Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Didrik Salve Galteland

Exploring Self-supervised Learning-based Methods for Monocular Depth Estimation in an Autonomous Driving Setting

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2021

NTNU
Norwegian University of Science and Technology

Didrik Salve Galteland

Exploring Self-supervised Learning-based Methods for Monocular Depth Estimation in an Autonomous Driving Setting

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

The idea of a neural network automatically learning the information we as humans want it to learn is the ultimate goal for deep learning due to labeling being both tedious and expensive. This thesis will show that, in specific fields, we are closer than ever to making this a reality.

Throughout this thesis, it will explore the inner workings behind how a neural network can automatically infer depth from only viewing a video from a single camera. After presenting the needed background knowledge, this thesis introduces self-supervised learning, what it is and how it can be trained to predict depth for a monocular video with no previous knowledge. When the required basic knowledge is established, the thesis continues by giving an in-depth explanation of some of the current state-of-the-art methods of doing self-supervised depth estimation, focusing on exploring how their contributions improved the field. These methods are validated through experiments, where they are tested on autonomous driving-focused datasets. The results from the experiments are discussed, where potential sources of error are presented together with potential fixes.

A brief analysis of the required modifications needed to use self-supervised depth estimation methods in an autonomous car is also presented, together with a reflection on the future of self-supervised depth estimation.

The thesis will present a potential novel architecture based on the findings and reflections. This architecture is based on combining features from all the current state-of-the-art self-supervised methods and has the potential to improve the current state-of-the-art.

Sammendrag

Ideen om å få nevrale nettverk til å automatisk lære seg det vi mennesker vil at de skal lære er det ultimale målet for dyp læring. Denne oppgaven vil vise at vi, for enkelte felt, nærmere enn noen gang for å gjøre dette til en virkelighet.

Denne oppgaven vil gjennomgående utforske de indre mekanismene som gjør det mulig for nevrale nettverk å predikere dybde ved å kun benytte seg av video fra et enkelt kamera. Etter å ha presentert den nødvendige basiskunnskapen vil denne oppgaven introdusere temaet "self-supervised learning"; hva det er og hvordan det kan brukes til å predikere dybde. Når basiskunnskapen er på plass vil oppgaven gi en mer grundig forklaring av noen av de toppmoderne metodene som benytter seg av "self-supervised learning" for å gjøre dybdeestimering, hvor oppgaven vil fokusere på hvordan disse metodenes bidrag forbedret feltet. Disse metodene er testet og validert i gjennom forsøk, hvor de blir trent og testet med dataset hentet fra autonome biler. Resultatene fra disse forsøkene blir diskutert, hvor potensielle feilkilder og mulige løsninger blir presentert.

Videre vil det bli gjort en analyse av hvilke endringer disse metodene vil trenge for å kunne brukes i en selvkjørende bil, sammen med en refleksjon av fremtiden til "self-supervised" dybdeestimering.

Oppgaven vil også presentere en potensiell ny arkitektur basert på resultatene observert i oppgaven. Denne arkitekturen er basert på å sette sammen deler fra de toppmoderne metodene til en ny arkitektur som har stort potensiale til å produsere resultater som kan overgå de toppmoderne metodene.

Preface

This thesis is a part of the NTNU Autonomous Perception Lab (NAP-lab) research group at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors Frank Lindseth and Gabriel Kiss, for providing access to the required resources and for their and guidance and feedback when writing this thesis. I also want to thank Paul Erik Frivold and Thomas Drabløs Frøysa for taking their time to help me proofread this thesis.

Thank you.

Contents

Figures

Tables

Acronyms

ANN Artificial Neural Network. 6–8

CNN Convolutional Neural Network. 5, 11, 15, 24–26

GAN Generative Adversarial Network. 25, 34

ILSVRC ImageNet Large Scale Visual Recognition Challenge. 11

ML Machine Learning. 5, 16

MSE Mean Squared Error. 8, 9

NLP Natural Language Processing. 15

ReLU Rectified Linear Units. 6, 11

RNN Recurrent Neural Network. 24

SfM Structure from Motion. 23

SotA State-of-the-Art. 5, 11, 12, 15, 24, 25, 28, 31–33, 39, 64, 83, 87

SSL Self-supervised Learning. 28, 29, 31, 32, 34–36, 42, 43, 58, 64, 68, 70, 74, 78–81, 87

Chapter 1

Introduction

1.1 Background and Motivation

Knowing the distance to other objects is a fundamental requirement for autonomous vehicles. Traditionally, this has been done by utilizing proximity sensors such as radars and ultrasonic sensors, and in the more modern times, lidars. The lidar creates the most accurate representation of the world around it by creating a highly accurate three-dimensional point cloud. But, a significant problem with lidars is their price and size. This has led companies like Tesla to already move away from using lidars. However, due to knowing the proximity of other objects being a fundamental requirement, these companies needed to look into other means of getting distance information. There are several promising approaches, but one of the more successful is using video from a single cameras to infer a depth value for all pixels in the image.

There are essentially three ways of generating depth from a video segment. The first method utilizes structure from motion techniques, with the most common being multi-view stereo techniques. These techniques utilize more traditional computer vision techniques and are based on triangulating the position of objects in the world. However, these methods can, at best, create sparse 3D reconstructions of a scene and often miss smaller significant objects like pedestrians. The second method uses a standard deep learning approach by automatically annotating the depth for images using onboard proximity sensors like radars. A significant problem with this approach is that to get this data, one would already need these expensive sensors, making the need for creating a system based on cameras irrelevant. Also, these sensors can only sparsely annotate the image due to the proximity sensors not being able to find depth for every single point in an image. This solution would also require an associated radar pointing in the same direction as the camera for every camera on the vehicle.

The third and relatively recent method of detecting depth is through self-supervision. These methods do not require any form of labels, as the neural networks automatically extract supervision signals from the data itself.

The performance gap between the supervised and self-supervised methods is shrinking fast, and with novel methods and ideas being presented at an alarming speed, it is not unsaid that this performance gap will be closed relatively soon. However, with new papers being published multiple times per month, this can make it hard to keep up with all their novel approaches and

contributions. This rapid progression makes it hard to understand what is needed to improve the field and how one can use the strengths of each novel contribution together.

Therefore, this thesis's primary goal will be to explore and investigate the current state-of-the-art self-supervised dense depth estimation, focusing on how these methods' contributions function and improve the field. However, to understand how the current state-of-the-art improved the field, one must first take a deep dive into the basics of self-supervised depth estimation. With this gained knowledge, one has the best possible position to be able to improve the field.

In order to validate the results achieved by these state-of-the-art methods, the thesis will also validate the results achieved by the methods by training them with different well-known autonomous driving-focused datasets, which differ from the ones that the methods used in their papers. This thesis will also analyze how the methods behave when using raw footage from a camera on NAP-lab's autonomous platform due to the other datasets being strictly curated for autonomous vehicles. Some thoughts on what is needed to incorporate these methods into an autonomous vehicle are also presented.

Finally, a novel architecture is proposed based on the findings after exploring how the current state-of-the-art methods improved the field. This architecture is based on using the contributions that improved the current state-of-the-art methods, where each element in the proposed architecture aims to improve shortcomings that the contributions by themselves introduces.

1.2 Research Questions and Goals

AsThe overall goal of this thesis is to gain the knowledge needed to suggest a novel architecture for self-supervised depth estimation that has the potential to improve the current state-of-the-art performance. This architecture should also be usable in an autonomous driving-setting, by being able to run in real-time. If successful, this architecture will potentially close the gap between self-supervised and supervised methods even further and is a step closer to removing expensive proximity sensors like lidars. To be able to propose a suitable architecture, some subgoals are proposed:

Subgoal 1: Gain the required knowledge about the fundamentals of self-supervised depth estimation.

Subgoal 2: Investigate and explore the current state-of-the-art self-supervised depth estimation methods, with a particular focus on how their contributions improved the field.

Subgoal 3: Select two of the state-of-the-art methods and train them on acknowledged datasets for autonomous vehicles. Also, select a state-of-the-art supervised method and compare the performance between the self-supervised and supervised.

A set of research questions related to the primary and subgoals are proposed to help guide the research. These will be answered in Chapter 5 in Section 5.6

RQ 1: Can self-supervised-based dense depth estimation methods achieve the same performance as supervised methods?

RQ 2: Can dense depth estimation methods replace proximity sensors like lidars and radars?

RQ 3: Which measures can be taken to improve existing state-of-the-art self-supervised methods?

1.3 Contributions

This thesis's main contribution is taking deep dive into the current state-of-the-art self-supervised dense depth estimation methods proposed in the last couple of years and evaluating these on unseen data. These findings can help to support decisions regarding replacing proximity sensors with a dense depth estimation system. Another contribution is the proposal of a novel architecture that has the potential to improve the current state-of-the-art performance. This architecture will be constructed by carefully selecting the contributions from the current state-of-the-art methods that have improved the field of self-supervised depth estimation.

Summarized, these are the thesis's main contributions:

1. **A in-depth analysis of the current state-of-the-art self-supervised depth estimation methods.**
2. **A proposed novel architecture that has the potential to improve the current state-of-the-art performance for self-supervised depth estimation.**

1.4 Thesis Structure

This section describes the layout of the thesis's chapters, where each chapter has a short description of its content.

Chapter 1: Introduction: Introduces the thesis and the topics it wants to study.

Chapter 2: Background and Related Work: Introduces relevant theory and knowledge, focusing on machine learning, self-supervised learning, and depth estimation.

Chapter 3: Methodology: Describes which depth estimation methods have been selected and their inner workings, and how they were modified to work with custom datasets. A description of how the selected datasets were organized, extracted, and preprocessed is also available in this chapter.

Chapter 4: Experiments and Results: Defines the experiments based on the research questions and covers the results of each experiment.

Chapter 5: Discussion: Analyses the results of the experiments and compares the results between each other. Some of the shortcomings and weaknesses are also discussed.

Chapter 6: Conclusion and Future Work: Summarizes the work done in this thesis and introduces some ideas for future work based on the knowledge gained.

Chapter 2

Background and Related Work

This chapter covers the background material and theory relevant to the thesis. It starts by introducing basic concepts in Machine Learning (ML). Following is an introduction to more advanced concepts like Convolutional Neural Network (CNN) and Transformers. Due to one of the main focuses of this thesis being Self-supervised learning, this topic is introduced thoroughly. Another significant part of this thesis is depth estimation, focusing mainly on self-supervised techniques to generate dense depth images. Here, some important works done in the field of dense depth estimation are presented, together with the current State-of-the-Art (SotA). Finally, the chapter introduces some datasets containing data from different autonomous vehicles.

2.1 Deep Learning

Deep learning is the foundation for most modern machine learning. It is based on artificial neural networks consisting of two or more hidden layers that can learn non-linear functions describing relationships found in data. The learning is done through backpropagation to minimize a loss function by using a strategy selected by an optimizer. The most used network types in computer vision are convolutional and recurrent neural networks, as these can utilize spatial and temporal features, respectively. This section explains further details of the concepts which make deep learning possible.

2.1.1 Building Blocks

2.1.1.1 Artificial Neuron

An artificial neuron is the building block of modern machine learning and is a simplified and mathematical interpretation of biological neurons found in brains. The overall concept is to take some input values, decide how important each input is, sum it, add a *bias*, and send it through an *activation function* to get the output called the *activation*. An illustration of a single neuron is shown in Figure 2.1. Mathematically a neuron calculates a value z that multiplies n inputs $\mathbf{x}$, with n weight values $\mathbf{w}$, added with a constant bias value b:

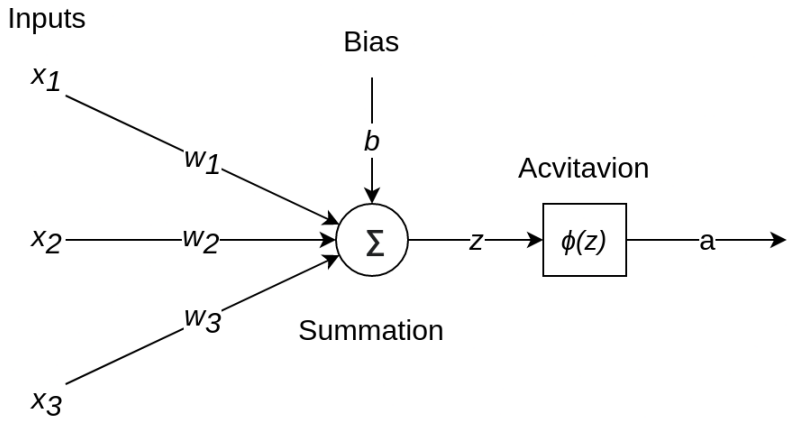


Figure 2.1: The inner workings of a neuron.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{2.1}$$

2.1.1.2 Activation Function

An activation function $\phi(z)$ is used to introduce non-linearity to the neuron to approximate non-linear and linear functions. It can also be used as a measure to limit the output z from Equation (2.1) between 0 and 1. The output a from an activation function $\phi(z)$ is considered the final output or activation of a neuron.

There exist multiple different activation functions. These are some of the most well known:

Sigmoid Historically, the sigmoid function $\sigma(z)$, shown in Equation (2.2), has been an important activation function. The sigmoid function limits an input to be between $[0, 1]$ in a non-linear fashion. Today, its mostly used in the output layer to output a probability value.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

ReLU The Rectified Linear Units (ReLU) activation function was introduced by Nair and Hintonn in 2010 [1] and is the most used activation function in modern machine learning. The function discards all values that are less than zero and outputs the value itself if it is greater than zero. The ReLU function used in an ANN increases both how quick the ANN learns and its ability to generalize. The ReLU function is shown in Equation (2.3)

$$R(z) = \max(0, z) \tag{2.3}$$

Softmax The softmax activation function transforms a layer of neurons output to a probability distribution, where the sum of all activations will equal one. This activation function is the

most used activation function for the output layer for classification problems, as the output represents a probability for the different classes. It is shown in Equation (2.4)

$$S(z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{2.4}$$

2.1.1.3 Artificial Neural Networks

Multiple artificial neurons can be connected to form an Artificial Neural Network (ANN). The goal of an ANN is to approximate an arbitrary function. ANNs consist of layers, which consist of at least one neuron. The first layer is considered the input layer, where the input to the neurons is the input data itself. The final layer is the output layer, which is the output of the network itself. The layers between the input and the output layers are called hidden layers, as the values of these layers are not visible in the output of the network.

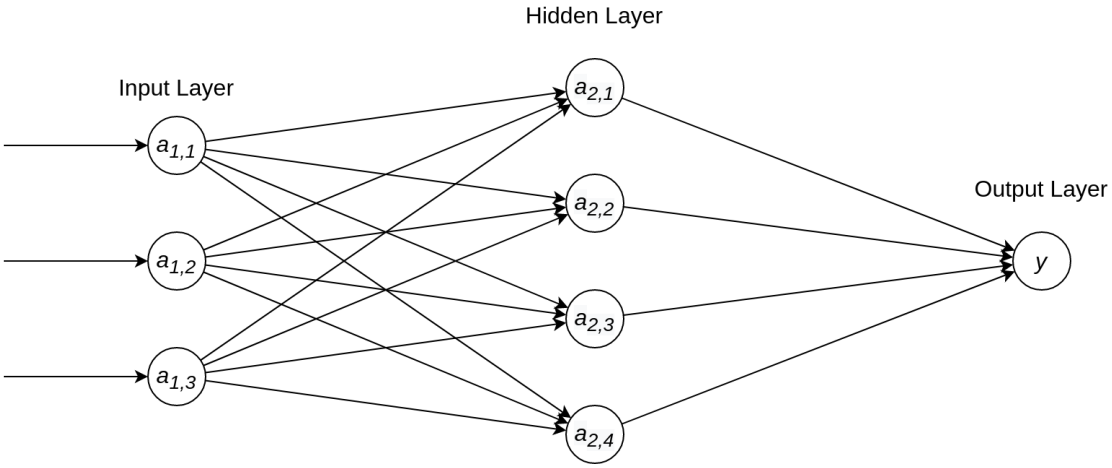


Figure 2.2: A neural network. The network consist of an input layer with three inputs, a hidden layer consisting of four neurons, and an output layer consisting of one output neuron. Each neuron has its associated weights and biases, as shown in Figure 2.1

One of the most used ANN architectures is the feed-forward network. In this network, layers are connected so that they form a *directed acyclic graph*, where the output from layer L_n is connected to layer a layer $L_{n+j}, j \geq 1$. When a cycle exists in the graph, the ANN is considered a *recurrent neural network*.

2.1.2 Learning

A neural network learns by adjusting the weights and biases found in the neurons, as these elements affect the network's final output. When a network is trained, a training sample is sent through the network. The activation of the last layers and the ground truth is sent through a *loss function*, where the loss represents how good a network predicted the expected output. To make the network better at predicting the expected output, one needs to adjust the parameters that affect the final layer's activation to better match the ground truth.

2.1.2.1 Loss Function

A loss function estimates the error of the output of the network against the ground truth. The network's output is produced from sending data through the network and reading the output layer's activation. This process is called a *forward pass*, as data is passed in a forward manner through the layers. As the loss function describes the network's error, it is beneficial for this function's output to be as low as possible. Therefore, we will need to change the network's output to be as close as possible to the ground truth data.

There exist multiple types of loss functions that apply to different situations. Here are some of the most used:

MSE The Mean Squared Error (MSE), also called "L2" loss, is a loss function often used in regression type problems. The function measures the error between a ground truth value y_i and a predicted value $\hat{y}_i$. The errors are added up, and the result of the function is the average of these distances. In other words: It's the mean ($\frac{1}{n}\sum_{i=1}^{n}$) of the errors squared $(y_i - \hat{y}_i)^2$. It is shown in Equation (2.5)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.5}$$

Where y_i is the ground truth value, and $\hat{y}_i$ is the predicted value. In an ANN, the $\hat{y}_i$ represents the activation a_i in the output layer.

Categorical Cross Entropy Loss The Categorical Cross-Entropy loss function is used when measuring the difference between the probabilities of more than two classes. It is most commonly used in classification problems and where the activation in the penultimate layer is using a softmax activation function, which is shown in Equation (2.4). The loss function compares a vector $\hat{\mathbf{y}}$ which contains predictions and the ground truth vector $\mathbf{y}$. Vector $\mathbf{y}$ is a one-hot encoded vector, meaning that only a single element in the vector equals one, representing the correct class, and all others are zero. When dealing with probabilities and possibilities the sum of all possibilities must be equal to one, thus: $\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \hat{y}_i = 1$. The Categorical Cross-Entropy Loss function is shown in Equation (2.6).

$$\text{CCEL} = -\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i \tag{2.6}$$

Where y_i is the ground truth value, $\hat{y}_i$ is the predicted value, and n is the number of unique classes. In an ANN, $\hat{y}_i$ is the activation a_i of the neuron in the output layer and represents a unique class's probability.

2.1.2.2 Backpropagation

Backpropagation is an algorithm used to figure out how sensitive the loss function is to changes in the different parameters found throughout a network. Imagine a neural network with k layers, where the first layer is the input layer, the last is the output layer, and the $k-2$ found between these are the hidden layers. All the layer only has one associated neuron for simplicity.

As discussed earlier, the last layer's activation is considered to be the output of the network. When a forward pass with a data point and ground truth is performed, one is left with the network's error. The goal of learning is to minimize this loss function. As both the input data and ground truth are immutable, the only way of changing the result of the loss function is to change the last neuron's activation.

A single neuron's activation in layer $n, 2 < n \leq k$, found in our imagined k layered neural network, is Equation (2.1) ran through an activation function $\phi(z)$. Equation (2.1) consists of the weight value w_n, the bias b_n, and the neuron's input x_n. Here, the value x_n will be activation a_{n-1} from neurons in previous layers in all layers except the input layer. As the activation a_{n-1} is determined by w_{n-1}, the bias b_{n-1} and the input x_{n-1}, only the weight w_n, and bias b_n value is directly tunable at this point. The only way to change $a_{n-1,i}$ is to change the weight and bias values found in the neurons in layer $n-1$. This pattern of changing a neuron's weights and bias so that the neurons connected to its output will change their activation will continue, recursively moving backward through the network until the input layer as the input to the network is immutable.

As one wants to change the weights and biases found in the layers, it is interesting to determine how much adjusting the weights and biases affect the loss function. The change of the loss function with respect to the weights found in the output layer can be expressed as:

$$\frac{\partial L}{\partial w_k} \tag{2.7}$$

Where, L is the loss function, w is the weight value of layer k.

It was previously determined in Section 2.1.2.1 that the cost is the difference between the final activation and the ground truth y. Section 2.1.1.2 discussed that an activation a equals the output of Equation (2.1) ran through an activation function $\phi(z)$. Therefore, one knows that the change in loss L is dependent on change in the activation a_k, which is depends on the change of the output z_k which all depends on the change of w_k. This means that Equation (2.7) equals:

$$\frac{\partial L}{\partial w_k} = \frac{\partial z_k}{\partial w_k}\frac{\partial a_k}{\partial z_k}\frac{\partial L}{\partial a_k} \tag{2.8}$$

This is the chain rule in practice, and the resulting equation gives us a specification of how much change in z affects change in the loss function L.

These partial derivatives can be calculated from the previously listed equations. Here, MSE is used as the loss function, and Sigmoid is used as the activation function:

$$z_k = w_k x_k + b_k \qquad a_k = \sigma(z_k) \qquad L = \frac{1}{2}(y - a_k)^2$$

$$\frac{\partial z_k}{\partial w_k} = a_{k-1} \qquad \frac{\partial a_k}{\partial z_k} = \sigma'(z_k) \qquad \frac{\partial L}{\partial a_k} = (y - a_k)$$

This results in

$$\frac{\partial L}{\partial w_k} = a_{k-1} \cdot \sigma'(z_k) \cdot (y - a_k) \tag{2.9}$$

For the bias, the only change needed is to use the partial derivative of z_k with respect to b_k, which equals 1. The change in the loss with respect to the bias is:

$$\frac{\partial L}{\partial b_k} = 1 \cdot \sigma'(z_k) \cdot (y - a_k) \tag{2.10}$$

In Equation (2.9) the term a_{k-1} refers to the activation found in the previous neuron. Here, the idea of propagating backward comes in, as one can figure out how much the change in terms found earlier in the network change the output of the loss function. The process of figuring out these partial derivatives is called backpropagation and is the core idea of how neural networks learn. Knowing how much the loss of how sensitive the loss function is to changes in the weights and biases is essential when the weight and bias values are tweaked to minimize the loss function. [1]

2.1.2.3 Optimizer

An optimizer describes the strategy used to change the values found in the neurons. It utilizes the gradients found in the backpropagation to find a new set of weights that minimizes the loss function. There exist multiple approaches and strategies for updating the weights. These are some of the most used optimizers:

Gradient Descent Gradient Descent calculates the new weight based on the gradients and the current weights. The algorithm considers all data points when calculating the new weights, resulting in relatively significant changes. An alternative to using all data points is using only one at a time, which is done in *Stochastic Gradient Descent*. However, updating after seeing a single example may lead to noisy changes, mainly if it contains some unusual data. One method of combating this is the *Mini-Batch Gradient Descent*, which calculates the new weights based on a batch of n data points. The size of a batch n is a static parameter specified before a training process starts. A weight update for a single data point, for the neuron in layer n in the imagined neural network, used in stochastic gradient descent is shown in Equation (2.11)

$$w_n^{t+1} = w_n^t - \alpha \cdot \frac{\partial L}{\partial w_n^t} \tag{2.11}$$

Where w_n^{t+1} is the new weight at time $t+1$, w_n^t is the current weights, and α is the learning rate

Adaptive Learning Rate Optimizers Adaptive Learning Rate Optimizers was first introduced by Duchi *et al.* [4] The idea with this class of optimizers is to adapt the learning rate as the updates of weights is done. This adaptable learning rate affects multiple dimensions and results in a training process where the learning rate does not need manual tuning. The

[1] It is recommended to watch 3Blue1Browns video on the topic [2, 3]

adaptability is introduced in a term G shown in Equation (2.12). G is the sum of the gradients squared up to time t. A small value ϵ is added to prevent division by zero. G is monotonically increasing, meaning that the learning rate tends towards zero and no further learning. AdaDelta, which was introduced by Zeiler *et al.* [5], introduces a term that prevents G from zero. Kingma *et al.* [6] wanted to improve this even more by introducing momentum in the form of Adam, which is considered the best general optimizer at the moment.

$$w_n^{t+1} = w_n^t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \cdot \frac{\partial L}{\partial w_n^t} \tag{2.12}$$

$$G_t = \sum_{j=1}^{t} \frac{\partial L}{\partial w_n^j}^2$$

2.1.3 Convolutional Neural Networks

A neural network containing convolutional layers is considered a Convolutional Neural Network (CNN). It was first introduced by LeCun *et al.* [7], and is considered one of the most important neural network types due to its ability to utilize sparse spatial features found in data structured into a grid-like pattern, e.g., images. A convolutional layer consists of multiple steps; a convolutional operation, an activation, and a pooling operation. The convolution operation generates multiple feature maps that are sent are sent through an activation function, typically a ReLU function. The last step is to perform pooling, which extracts features from the data and effectively shrinks its dimension. Using convolutional layers drastically reduces the number of parameters found throughout the network compared to fully connected layers.

2.1.4 Backbone Architectures

There are many ways of arranging different types of layers in a neural network, and some are better and others. Here are some of the more historically important architectures that significantly improved the previous SotA:

2.1.4.1 AlexNet

AlexNet is a CNN architecture that was created by Krizhevsky *et al.* [8] in 2012. It was an entry to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which significantly outperformed all previous entries. It quickly became a staple for using CNNs and Deep Learning in image applications, as it was the use of convolutions that made it superior to, e.g., only using fully connected layers.

2.1.4.2 VGG

VGG is a CNN model proposed by Simonyan *et al.* [9] and was an entry to ILSVRC in 2014. It is an improvement of AlexNet mostly due to its utilization of a deep architecture and smaller filters in the convolutional layers. However, it is still relatively slow to train due to the numerous parameters found in the layers.

2.1.4.3 ResNet

ResNet was created by He *et al.* in 2015 [10] and was considered the most groundbreaking work since AlexNet. It introduced identity shortcut connection through residual blocks, making it possible for data to skip between layers without running through convolutions and activation. ResNet and residual blocks have been the base and inspiration for newer architectures such as Inception-ResNet [11].

2.2 Transformers

A transformer is a deep learning architecture introduced in the 2017 paper "Attention is All You Need" by Vaswani *et al.* [12]. It is an architecture consisting of an encoder and a decoder, where the attention mechanism plays a crucial role. Transformers have quickly become the base for multiple SotA methods of processing sequences [13–16].

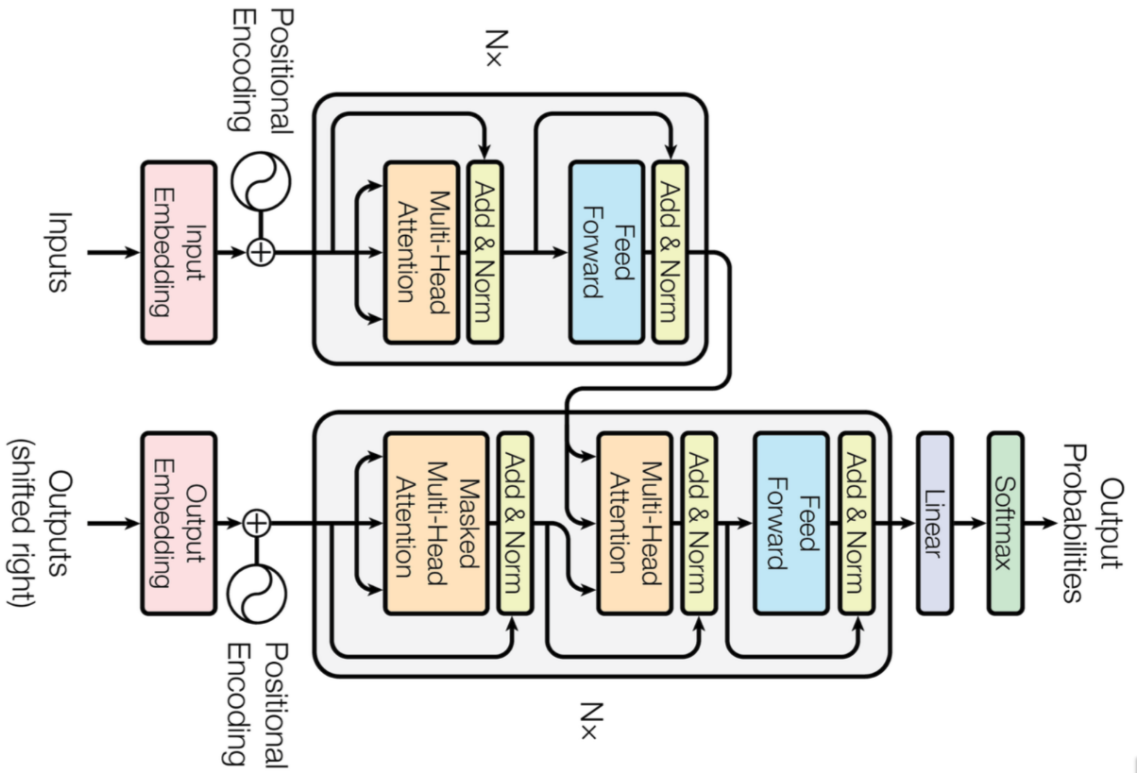


Figure 2.3: The architecture of a transformer. Adapted from Vaswani *et al.* [12]

This subsection will introduce the different components of a transformer. Due to transformers being initially created for NLP tasks, this subsection will use examples related to NLP when explaining the different parts of the transformer.

2.2.1 Encoder

2.2.1.1 Input Embedding

Data needs to be translated into a format that a computer can understand, preferably numbers, before being used in any form of ML-based system. Unfortunately, transformers are primarily used in NLP cases where the input consists of words and sentences, which on their own are not easily representable by numbers. Transformers fix this by using embedding spaces, which maps words to a vector, representing a point in space. This mapping maps words with similar meanings close to each other in the embedding space. The embedding space can either be learned while training or be pretrained.

2.2.1.2 Positional Embedding

A word can have different meanings depending on its position in a sentence. Therefore, transformers need information about a word's context in a sentence. This context is assigned with a positional embedding that encodes the word's position in a sentence to a vector.

2.2.1.3 Attention

Attention is one of the novel ideas that made transformers superior to existing architectures. The idea of attention is to establish which parts of the input the model should focus on. The attention is captured by creating a vector for each word that describes how much the "i-th" word in the sentence relates to the other words with a numerical value. One problem here is that the attention vector for a word also contains itself and tends to overestimate its relation with itself. This overestimation is fixed using multi-headed attention, which generates multiple versions of the attention vector for each word and uses a weighted average for its final attention vector.

2.2.1.4 Feed-forward Net

At the end of the encoder, there is a standard fully-connected feed-forward network that uses the attention map as input. In practice, this network is used to transform the output from the attention block to a format that either another encoder block or the decoder block can use. It is worth noting that multiple feed-forward nets are created, specifically one for each attention vector, so that the transformer can input multiple words simultaneously.

2.2.1.5 Encoder Architecture

The prementioned components are put together to form the encoder. The encoder is split into steps only happening once and steps that can be repeated. The non-repeatable portion consists of the input embedding and positional encoding, as once a sentence is translated into vectors, there is no need to repeat this step. The repeatable portion consists of multi-headed attention blocks and feed-forward nets. This portion can be repeated multiple times, with the input and output being a set of vectors. The final output of the encoder block is a set of vectors that is used as input to a decoder block. The complete architecture of a basic transformer can be seen in Figure 2.3

2.2.2 Decoder

2.2.2.1 Embedding

During the learning phases, the target is fed to a decoder. As in the encoder, this target needs to be embedded in an embedding space. However, the embedding space used for the input and output is separate from each other due to the context of a word being encoded might not be equal to ground truth. An example is when a transformer is used in a translation task. Here, one word in one language might not have the same context as in another language.

2.2.2.2 Masked Attention

The masked attention block works similarly to the attention block in the encoder. The main difference between these is that in the masked attention block, for a word at position i, the attention is only calculated for the previous words in the sentence. For all the words at a position greater than i, the attention value is set to 0. This masking is done to prevent the transformer from having the target available during training. Without it, the transformer would learn to output the following word in the target and not learn anything about the actual relationship between the input and target data.

2.2.2.3 Encoder-decoder Attention

The second attention block in the decoding portion is called the encoder-decoder attention block. This block uses the output from the encoder and the output, containing the vectorized input data, and the output from the masked attention block, containing the vectorized target data. With this data available, this block determines how related each attention vector is to each other and is the primary source of learning the relationship between the input and the target.

2.2.2.4 Decoder Architecture

The architecture of the decoder block also consists of non-repeatable and repeatable parts. Following the decoder, the non-repeatable part consists of input embedding and positional decoding. This input is fed to the repeatable parts of the decoder, which consists of the masked attention block that outputs to the encoder-decoder attention block, together with the output from the encoder.

For each training step, the transformer tries to predict the next word in the sentence provided in the target. This process is repeated until the transformers predict that the sentence is finished with an "end-of-sentence" token.

The final output from the decoder is fed into a final feed-forward network. This network is primarily used to expand the number of outputs to the number of words in the language of the target language. A final softmax activation function maps the output to a probability distribution.

2.2.3 Vision Transformers

Convolutional Neural Networks has for many years been dominating the image recognition field ever since the launch of AlexNet [8] in 2012. The paper "An Image is Worth 16x16 Words" by Dosovitskiy *et al.* [17] showed that the success transformers have had in NLP tasks also can be utilized in a computer vision setting. They showed that a transformer-based architecture could achieve close to SotA CNN-based methods by using novel ideas for preprocessing images into a format that a transformer could use.

This section will introduce some of these novel ideas and how the images were preprocessed to be used in a transformer.

2.2.3.1 Patching

While a sentence typically consists of a reasonable number of words, a single image can consist of many millions of pixels. While CNN networks like ResNet have little to no problems using images of size 250x250, this quickly becomes a problem for transformers when calculating attention, as attention is a quadratic operation. Trying to calculate attention for a single pixel will result in a vector of 250^{2^2} size, which is impossible to calculate with today's hardware.

CNNs solve this problem with the convolution operation, which reduces the image dimension into features, effectively creating a larger and larger receptive field until one has close to a global receptive field in the last layers. Due to the original transformer attending to every part of the input in a single pass, the visual transformers also wanted to have a way to attend to the entire image globally.

The visual transformer solves this by using global attention by using image patches of size 16×16 instead of using standalone pixels. Using patches fulfills the transformer's requirement of using a set as its input. Following the original transformer, these patches are encoded with a positional value representing the original patches' position.

2.2.3.2 Class Token

In order to be able to perform classification tasks, the visual transformer has an extra learnable embedding that represents the target class. This token is also fed into the transformer.

2.2.3.3 Prediction

After the data has been embedded, encoded, and has a classification token added, this data is fed to a standard transformer encoder-decoder architecture. However, the final output head from the transformer is replaced with a single feed-forward network that only has a connection to the classification token. The rest of the output from the transformer decoder is discarded. This output is run through a softmax activation function and functions as a standard classification in a feed-forward network. The final output is a probability distribution for which class the input image represents.

2.3 Machine Learning Approaches and Branches

There exist multiple approaches to solve tasks with Machine Learning (ML). This section is dedicated to introducing the relevant approaches to this thesis.

2.3.1 Supervised Learning

Supervised learning is an approach to machine learning that utilizes data with associated, human-annotated labels. It is defined by having a dataset **D** consisting of data **X** and associated labels **Y**. In a dataset **D**, a data point $x_i \in \mathbf{X}$ has an associated label $y_i \in \mathbf{Y}$. Loss for a batch $d \in \mathbf{D}$ [18] consisting of n examples of labeled training data $d = \{x_i, y_i\}_{i=0}^{n}$ is defined as:

$$loss(d) = \min_{\theta} \frac{1}{n} \sum_{i=1}^{n} loss(x_i, y_i) \tag{2.13}$$

Where the $loss$ function predicts the difference between the ground truth y_i and, the predicted label $\hat{y_i}$ from x_i. θ is the weights. Loss is further explained in Section 2.1.2.1

2.3.2 Semi-supervised Learning

Semi-supervised learning focuses on learning using a sparse amount of labeled data while simultaneously utilizing a broad amount of unlabeled data. The methods combine supervised learning with unsupervised learning and categorize it as something in between the two. The two main methods in semi-supervised learning are transductive and inductive learning.

Having n data examples consisting of $x_1, ..., x_n \in \mathbf{X}$, labels $y_1, ..., y_n \in \mathbf{Y}$ and $x_{n+1}, ..., x_{n+u} \in \mathbf{X}$ unlabeled examples, the goal of transductive learning is to infer the correct labels for the unlabeled data $x_{n+1}, ..., x_{n+u} \in \mathbf{X}$. Inductive learning takes it a step further by also trying to produce a classifier from the unlabeled data. It is transductive learning that is most used in computer vision and machine learning use-cases.

2.3.3 Weakly-supervised Learning

Weakly-supervised Learning is Supervised Learning that utilizes labels categorized as weak labels. Labels are defined as weak when they are:

- Sparse, as accurate labels are hard or not possible to obtain. e.g., in novel use-cases
- Inaccurate, as it is collected from sources that are not necessarily quality controlled, e.g., from a crowdsourced dataset.
- Lacking all relevant and usable information for the given task, e.g., having labels describing categories when developing an object detection task.

Loss for weak-supervised learning is defined equal to Equation (2.13), with the only difference being **Y**, now consisting of only weak or a combination of weak and accurate labels.

2.3.4 Unsupervised Learning

Unsupervised learning refers to an approach that does not utilize any preexisting, human-annotated data as supervision during training. The learning methods in unsupervised learning are based on identifying patterns and information directly from the data, with cluster analysis being one of the more known techniques.

2.3.5 Self-supervised Learning

Self-supervised learning is a branch of unsupervised learning. The goal of self-supervised learning is to automatically generate a supervision signal from the data itself that can be used to solve a task. The generated supervision can be used for specific tasks or used as pre-training as an alternative to pre-trained weights generated from human-annotated labels.

The idea of not using humanly annotated labels is one of the key topics for this thesis. Therefore, the thesis will give a more in-depth explanation of the topic.

2.4 Self-supervised Learning

Self-supervised learning uses the existing structured information to learn features and patterns typically learned by supervised learning. These features and patterns are found by training the model on *pretext tasks*, which can be done by transforming or augmenting the image and then using an ML model to predict the transform or augmentation. The core idea of self-supervised learning in computer vision is that a model will need to learn spatial information from the data to correctly predict the transform or augmentation. After a model has been trained on a pretext task, the learned knowledge can be transferred to a more useful task and validated with a *downstream task*, which can be any easily measurable task that utilizes the same input as the pretext task. These tasks are often tasks found in typical supervised learning approaches, as there exist several pre-trained models for these kinds of tasks. Figure 2.4 shows the standard pipeline of a self-supervised learning system.

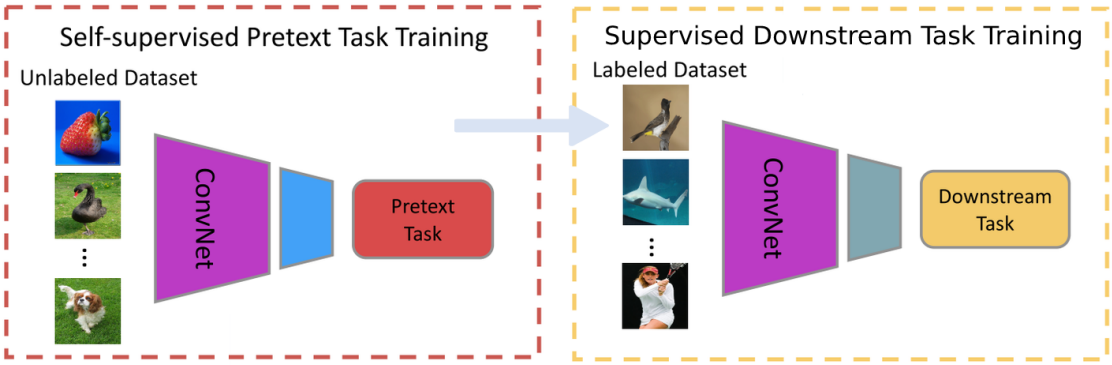


Figure 2.4: Pipeline of a self-supervised learning approach. Adapted from Keshav *et al.* [19]

2.4.1 Pretext Tasks

The idea of pretext tasks in self-supervised learning is to occlude some information found in the data and generate a model that can learn the occluded data. Pretext tasks can also include augmenting the data, with the model's objective being to learn the augmentation. The data itself will always generate the supervision signal, with no human intervention at any point.

The pretext task aims to make the model learn features and patterns found in the data, as one usually would do with a supervised signal. The need for humanly annotated data is reduced when pretext task training is utilized, as the model trained on the pretext task can be used as a base for future fine-tuning on downstream tasks.

There exist many different pretext tasks that all have some strengths and weaknesses. The pretext tasks listed below are some of the more common and proven pretext tasks.

2.4.1.1 Rotation

One of the most popular pretext tasks is rotation. Here, an image is rotated *0*, *90*, *180*, or *270* degrees before it is sent to a model. The model's goal is to predict the rotation applied to the image with a 4-way classification task. The rotation pretext task is proven to work empirically [20, 21], as the model needs to understand patterns and features represented in the image to predict the rotation correctly. An example of possible rotations for the pretext task is shown in Figure 2.5

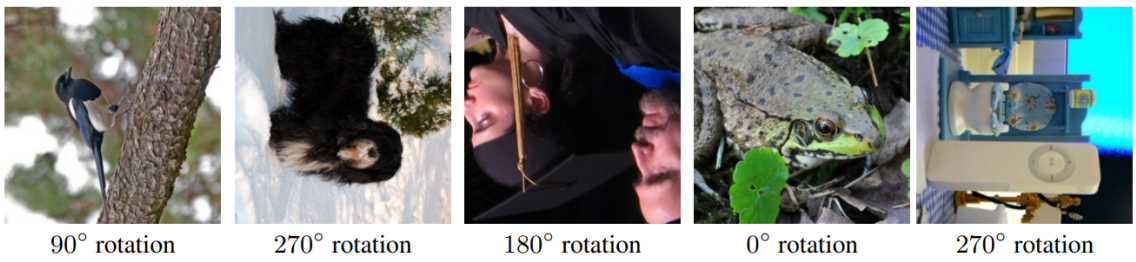


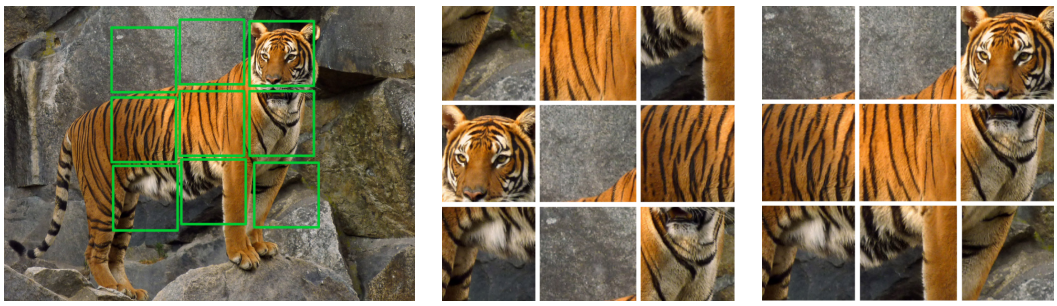
Figure 2.5: Possible rotations that the pretext task utilizes. Adapted from Gidaris *et al.* [20]

2.4.1.2 Jigsaw Puzzles

Jigsaw puzzle pretext tasks consist of taking a cropped part of an image, splitting it into an NxN grid, and finally shuffle the grid. The NxN image pieces are sent through a model that needs to predict the order of the image pieces so that the result is the unedited version of the cropped image. The steps in the pipeline are shown in Figure 2.6.

2.4.1.3 Colorization

The colorization pretext task consists of removing the color channels from an image and training a model to predict the missing channels' values. Both Larsson *et. al.* [23] and Zhang *et. al.* [24] prove that colorization is a powerful pretext task. Specifically, Larsson *et. al.* shows that



(a) **(b)** **(c)**

Figure 2.6: The jigsaw pipeline. (a) shows the selected image tiles from the full image. **(b)** shows the shuffled tiles, and **(c)** shows the image tiles in a correct predicted order. Adapted from Noroozi *et al.* [22]

colorization can have the same results as using a pre-trained model with annotation from ImageNet as a base model for downstream tasks. Downstream tasks are discussed in Section 2.4.2.

Figure 2.7: Images colored by a colorization model. Adapted from Zhang *et al.* [24]

2.4.1.4 Pretext task for video: Temporal Order Verification

There also exist pretext tasks for video. One of these is temporal order verification, which is the idea of verifying that a sequence of image frames is in the correct order and not shuffled. The core idea is that one selects N number of frames, where all frames are in order, shuffles the frames in a random order, and predicts the frames' temporal order. Misra *et al.* [25] explore this in their paper covering unsupervised learning using the spatiotemporal signals found in videos of human actions.

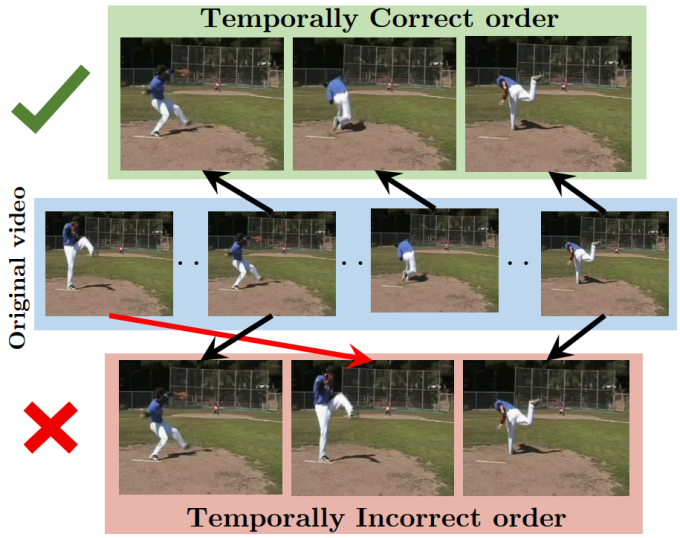


Figure 2.8: Description of temporal correct order. Adapted from Mistra *et al.* [25]

2.4.2 Downstream Tasks

Downstream tasks are applications used to evaluate the quality of the model that was trained with pretext tasks. These tasks primarily consist of tasks used in real-world applications, often found in typical supervised learning-based applications. The goal is to use the trained backbone network from the pretext task as a base for further fine-tuning. Some typically used backbones are found in Section 2.1.4. The fine-tuning is done by attaching a head network and training the network in a standard supervised setting. In a few cases, the downstream task may be equal to its pretext task, and so the head used in the pretext task is kept.

In computer vision, the most common downstream tasks are classification, object detection, and segmentation.

2.4.2.1 Classifying

Classification is the task of specifying a group or category that best describes a data point. Classification can be supervised and unsupervised, e.g., K-means clustering, although it will always be supervised when used as a downstream task. A typical downstream task in a computer vision setting is classifying the image's content into a single class.

2.4.2.2 Object Detection

Object detection is the task of finding objects of interest in an image and simultaneously classifying the found objects. The output for an object detection task is a bounding box describing wherein the image an object of interest exists, a class or category describing the object in the bounding box, and a confidence score describing how sure the network is that the prediction is correct.

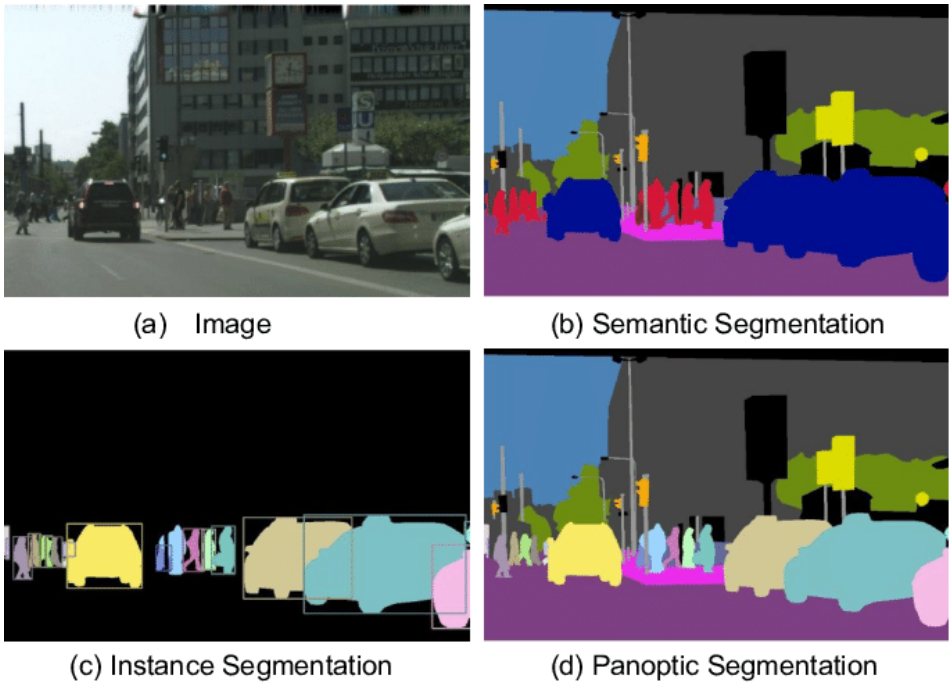


Figure 2.9: Different kinds of segmentation. Adapted from Chen *et al.* [26]

2.4.2.3 Segmentation

Segmentation is the task of clustering and classifying an image into regions of known classes. The output of semantic segmentation is the shapes that describe the different classes found in the image. There exist multiple subclasses of segmentation. The most basic is semantic segmentation, which classifies all pixels into a specific class. Instance segmentation classifies only the regions of interest and differs from different instances of classes. Regions that are not of interest are not classified. Panoptic segmentation combines instance and semantic segmentation.

2.5 Depth Estimation

Using depth data is a crucial component for autonomous vehicles, as they need to understand the environment around them to function optimally. There exist multiple approaches to extract depth information for autonomous vehicles. Traditionally, they have mainly used separate sensors for detecting depth. However, there has been a greater focus on detecting depth accurately using only images as input in later years. This focus originates from the wish to reuse existing sensors on the vehicle rather than having separate sensors. Range sensors can often also be a pretty costly addition to a sensors suite. The reasons stated are used as motivation by autonomous car manufacturers like Tesla to look into separate range sensors. This section explores the different possibilities for extracting depth information for an autonomous vehicle, primarily focusing on extracting depth from images.

2.5.1 Separate Sensors

Using separate sensors is one of the earliest and most common forms of retrieving depth for autonomous vehicles. This section briefly lists some standard sensors used on autonomous vehicles used to retrieve depth.

2.5.1.1 Radar

RAdio Detection And Ranging sensors are sensors that use electromagnetic waves to retrieve depth. The sensor sends out EM waves and measures the time it takes to receive a reflection of the signal. This time can be used to calculate the approximate depth of an object. Radars have an impressive range, but the resulting depth information has a relatively low resolution than misses out on objects' more refined details. The radar sensor also works in most weather conditions, like snow, rain, and hail, making it a reliable addition in a sensor suite for finding larger objects.

2.5.1.2 LiDAR

LIght Detection And Ranging are sensors that use light waves to retrieve depth. The sensor sends out light pulses and measures the time it takes for the pulse to return. The time used between the pulse and the returning signal can create a highly accurate 3D map of the world around the sensor. These maps are more accurate than Radar detections due to the light waves having a shorter wavelength. However, LiDARs may suffer in any weather conditions that have any form of precipitation. This can quickly become a problem in more arctic environments, where snow is common in the winter half of the year. Lidars are great sensors when looking for finer details in objects, or in general, smaller objects around a vehicle. However, they are currently quite costly and can be a challenge to interfere with due to the vast amount of data it produces.

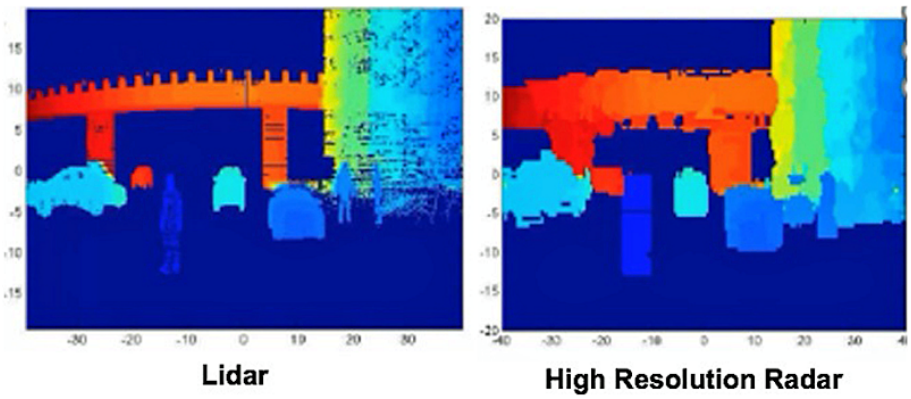


Figure 2.10: LiDAR compared to Radar depth image. Adapted from fierceelectronics.com

2.5.1.3 RGB-D Cameras

RGB-D cameras are specialized cameras that emit a speckle pattern of infrared light in the direction that it is pointing. The infrared speckle pattern projected onto objects in front of the camera is observed and is compared to a ground truth speckle pattern. Measuring the differences in the observed pattern and the ground truth pattern generates a disparity map used to determine depth. These cameras have a lower range but can produce very accurate depth maps for the distances in their operation window. They are primarily used in indoor settings.

2.5.2 Geometry-based Approaches

Geometry-based approaches use multiple images taken of the same setting where the images have slight changes in where they were taken. Extracting features from these images and comparing the features with each other. These are some of the most common ways of using geometry to extract depth.

2.5.2.1 Stereo Matching

Stereo matching is a geometry-based technique of detecting depth from two separate cameras pointing in the same direction. The core idea is to match pixels from one of the cameras with the other, measuring how the difference between the two pixels in each of the images generates a disparity value. Doing this for all matched pixels in the images generates a disparity map. The map is used to triangulate the camera's distance to the point in space represented by the pixels. Using this method requires that the cameras' baseline distance is known and not changing, calibrated cameras with a reasonable rectification of the images and that the cameras themselves are synced for optimal results.

2.5.2.2 Structure from Motion

Structure from Motion (SfM) is a geometry-based technique of detecting depth from a sequence of images. The technique utilizes the structural information gained by comparing features and differences found in a sequence of 2D images. The accuracy of the technique is based on accurate and consistent features found in the sequence. SfM also suffers from ambiguities in shapes as the camera moves [27].

2.6 Dense Depth Estimation from Images (Related work)

The idea of dense depth estimation from images, is to infer a depth D_t for all pixels in an image I_t from images $\boldsymbol{I}$ with a model θ_{depth}:

$$D_t = \theta_{depth}(\boldsymbol{I}) \tag{2.14}$$

$\boldsymbol{I}$ can either be a singular frame I_t, which is used in monocular depth estimation methods, or a set of frames $\{I_t, I_{t-1}, I_{t-2}, ..., I_{t-n}, \}$ used in multi-frame monocular depth estimation methods, or a set of frames $\{I_t^1, I_t^2, ..., I_t^n, \}$, where the superscript represents cameras from different angles, where typical setups often includes a stereo pair of images [28, 29].

The model θ_{depth} has utilized forms of learning introduced in Section 2.3 to learn how an image should be associated with depth. Supervised learning-based methods can utilize supervision signals about depth from other depth sensors, like lidar or radar scans. Another form of supervision is annotated data created by humans. These methods often use a single deep network for regressing depth from the supervision signals. SotA methods are often accompanied by some form of supporting technology that improves the predictions.

Methods using other supervision signals like the vehicle's ego-motion [30], stereo matching labels [31] or other forms of supervision signals that are not the actual depth information. These supervision signals can benefit the depth network or other supporting networks like the pose network in self-supervised approaches.

One of the more interesting approaches to learning-based depth estimation is precisely the use of self-supervised learning. These methods do not rely upon nor utilize any external supervision signals but generate their supervision signals from data found in the images. Currently, self-supervised methods are still being outperformed by supervised approaches. However, these methods have a significant advantage over methods that need explicit supervision signals due to their possibility to utilize data where these supervision signals are not available, e.g., on UAVs with no range sensors [32].

Dense depth estimation from images have a great number of use cases in autonomous vehicles [33, 34], AR/VR applications [35] and in medical applications [36]

This section will also function as a related work section due to this thesis covering different kinds of dense depth estimation methods.

2.6.1 Supervised Learning Approaches

As explained in Section 2.3.1, a supervised learning approach requires ground truth labels. In a dense depth estimation type of task, these labels are generally created by external range sensors, as it is hard for a human to give an accurate description of depth, especially in a 2D setting.

One of the first looks into supervised learning-based depth detection was done by Eigen *et al.* [37]. Their solution is based on having a CNN consisting of two components; one for detecting coarse depth on a global level and one for refining the coarse depth. The coarse depth image is fed into the refining part, together with the original input image, which produces more refined details and results in a new depth map. During training, an ground truth depth map is provided together with an RGB image. Their network directly regresses on the depth map using a loss function based on scale-invariant error. There exist other noticeable CNN-based solutions, e.g., with Laina *et al.* [38], which directly follows up Eigen *et al.*, proposing a deeper residual net for the task.

Another supervised approach is using a Recurrent Neural Network (RNN) to detect depth based on sequences of images. One notable approach here is done by Wang *et al.* [39]. Their contribution mainly focuses on multi-view image reprojection and forward-backward flow consistency

losses. Based on convolutional LSTM units, the model can be trained both in a supervised and unsupervised manner, where the supervised approach utilizes ground truth depth maps as a supervision signal, together with a sequence of ten images forward and ten images backward from time t.

There also exist methods that utilize Generative Adversarial Network (GAN) to generate depth maps. Lore *et al.* [40] proposes a conditional GAN for the task, which is a method of training GANs that adds conditional information to the generator and discriminator. The conditional GAN based system can be trained with three different approaches, utilizing a single monocular image, a sequence of monocular images and monocular images, and the accompanying optical flow patterns. With these inputs, they generate an estimated depth map representation. The conditional information used is gamma-corrected lidar maps with the same height and width as the input image.

2.6.1.1 Ada Bins

"*AdaBins: Depth Estimation using Adaptive Bins*" is a paper from 2020 by Bhat *et al.* [41] and is the current SotA paper on monocular depth estimation. The motivation behind the paper is that current architectures at the time of writing their paper did not perform enough global analysis of the output values. This problem is a known problem with CNNs, as one can only process global information about an image with a meager spatial resolution close to the end of a CNN. To better utilize this global information, Bhat *et al.* propose utilizing this information at a higher resolution. They do this with their novel AdaBins module, which uses a transformer-based architecture that divides the depth range into bins that center values estimated adaptively per image.

2.6.1.2 Dense Prediction Transformers

"*Vision Transformers for Dense Predictions*" by Ranftl *et al.* [42] takes the idea of using transformers for dense depth one step further by utilizing a transformer-only-based architecture. As AdaBins, Ranftl *et al.* also comment on the sole use of CNN-based encoder-decoder architectures in dense prediction and the drawbacks downsampling introduces for tasks that benefit from high-resolution images.

Ranftl *et al.*'s proposed improvement over the use of the convolutional block is to introduce a novel architecture; the dense prediction transformer. This architecture uses the vision transformer architecture, discussed in Section 2.2.3, as its backbone. While the input to vision transformers still undergo downsampling, the main difference between using vision transformers and using CNN is that the vision transformer will perform the embedding before the image is downsampled. Using a vision transformer also gives a global receptive field through attention, rather than using local receptive fields that a CNN effectively uses.

Rather than only generating monocular depth estimation, this dense prediction architecture can generalize to all dense prediction tasks, like semantic segmentation and more. Ranftl *et al.* have tested their novel architecture on both these tasks, and they achieve SotA results in both use cases.

2.6.2 Self-supervised Learning Approaches

The field of self-supervised dense depth estimation is still relatively new. However, there still exist some notable works did that changed the field when they were published. This section presents some of these notable works, together with some of the current state-of-the-art methods.

2.6.2.1 Monodepth

Monodepth was introduced by Godard *et al.* [28] in 2017 with a novel training objective that enabled a CNN to perform single image depth detection. The novel training objective consists of generating a disparity map from a single image by utilizing an image reconstruction loss. This approach requires a stereo pair of images to generate the ground truth disparity map during training, which is used as a supervision signal. The paper also showed that the accuracy could be boosted by performing the image reconstruction from both the left and the right image pair during training. The method yielded state-of-the-art results when it was introduced, outperforming current supervised methods.

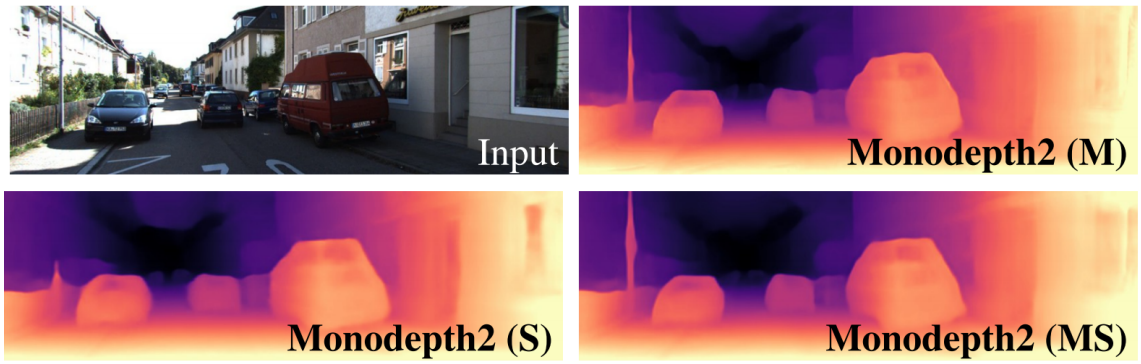


Figure 2.11: Depth detection from Monodepth and Monodepth2. The output is from three different models, trained with (M) monocular video, (S) stereo pairs, and (MS) a method combining the stereo pairs and the monocular video approaches. The output from Monodepth2 is noticeably improved compared to the original Monodepth prediction on the same image shown. Adapted from Godard *et al.* [29]

2.6.2.2 Deep Hints

Photometric reprojection losses are used within self-supervised learning systems for depth estimation, resulting in multiple local minima. Multiple minima can heavily restrict regression networks performing these tasks, with visual artifacts around thin structures being typical. Watson *et al.* [43] introduced a technique to combat this called 'Deep Hints'. This technique helps the network learn better weights and is calculated with the information already available. The hints themselves are complementary depth suggestions obtained from standard stereo algorithms and enhances the photometric reprojection loss. As the method uses stereo techniques for acquiring the deep hints, it requires stereo pairs of images as input.

2.6.2.3 Unsupervised learning of depth and ego-motion from video

Zhou *et al.* [44] presented a method of training both pose and depth networks simultaneously with monocular video as input. They use view-synthesis as supervision, meaning they try to create a new image of a scene taken from another position. Due to ego-motion from the vehicle they use to collect the data, they automatically obtain close to ground truth data for this task. Zhou *et al.* uses KITTI as their dataset and takes it a step further by excluding all static sequences found in the dataset, creating a new subset of KITTI. This subset is now used by other methods utilizing monocular video as input.

2.6.2.4 SfM-Net

SfM net is a geometry-aware neural network created by Vijayanarasimhan *et al.* [45] that predicts motion estimation in videos, depth, camera motion, and object rotations and translations all using the same architecture. The network can be trained both self-supervised and fully supervised by a known ego-motion or depth from RGB-D sensors. The depth is predicted in a separate branch of the network called the structure network, using a U-net encoder-decoder model, with a single frame as input and a point cloud as output. The point cloud is later fed into other parts of the network to extract the object flow. Their self-supervised training loss is based on minimizing the first and second frame photometric error, where the second is warped towards the first according to the predicted motion field.

2.6.2.5 Monodepth2

Monodepth2 [29] is the next iteration of Monodepth, which introduces new features and improvements to the existing Monodepth system. One significant new feature was introducing a training strategy that made it possible to train on monocular video as a standalone option to stereo pairs. A training strategy utilizing both options is also available and is the method yielding the best results. Other significant improvements were a new loss function to better handle occlusion, a new sampling method that reduces visual artifacts in the final result, and an auto-masking loss that ignores pixels that do not change when the camera moves in the monocular video. These improvements lead to Monodepth2 continuing to show state-of-the-art results compared to other solutions.

2.6.2.6 PackNet

"*3D Packing for Self-Supervised Monocular Depth Estimation*" was published in 2020 by Guizilini *et al.* [30]. At the time of publishing, the standard way of improving self-supervised learning methods was to engineer the loss function. Guizilini *et al.* argue that simply engineering the loss function has its limits and that there is a significant potential for increasing performance by changing the model itself. They introduced a novel convolutional network architecture called *PackNet* that improves the down and upsampling in the encoder-decoder architecture. This architecture utilizes their novel *Packing* and *Unpacking* blocks which use 3D convolutions. These blocks can fold the spatial dimension of a convolutional block's feature map into extra feature channels, which in contrast to striding or using pooling layers, are invertible with no

loss. This feature space is learned to be then compressed, and the unpacking block learns to expand this compressed representation, both by using 3D convolutions.

2.6.2.7 Feature Depth

Virtually all SSL-based depth estimation tasks utilize photometric error in their loss function. Shu *et al.* address the problems of only relying on this error estimation in a dense depth prediction task in their paper "*Feature-metric Loss for Self-supervised Learning of Depth and Egomotion*" [46]. They discuss that the photometric error function falls short in low-texture regions like walls, billboards, poorly lit areas, where the photometric error estimates an artificially low error, resulting in the methods predicting the wrong depth and pose when in reality the error can be significant. By introducing a novel idea of feature-metric loss, they can achieve the current SotA result for all SSL-based dense depth detection methods. The feature metric loss is based on a separate encoder-decoder network that generates features used when calculating the total loss for the architecture.

2.6.2.8 ManyDepth

Most SSL-based monocular dense depth estimation methods use multiple input frames during training, but only a single input frame when the system is used in testing. This means that important sequential information is available but not utilized at test time. The few implementations that use this information are either too computationally expensive or only use geometrical data. Watson *et al.* address these issues in their 2021 paper "*The Temporal Opportunist: Self-Supervised Multi-Frame Monocular Depth*" [47]. Their main contribution is *ManyDepth*, a novel architecture that utilizes cost-volumes to use multiple frames at test time. This novel architecture outperforms all earlier publications on SSL-based monocular dense depth estimation when utilizing a ResNet50 [10] backbone architecture and test-time refinement. The same team created Manydepth also created Monodepth and Monodepth2 and can be seen as the next iteration in the "Monodepth" class of systems.

2.7 Datasets

This thesis wants to explore dense depth estimation in an autonomous driving setting. As this thesis is written in collaboration with NAP-lab, it would be beneficial if NAP-lab data could be utilized here. However, as data from the NAP-lab became available only a week before the thesis's due date, publicly available datasets will be used as the primary training datasets for this thesis. The following section introduces some of these many publicly available datasets that can be used for dense depth detection. The requirements for these datasets are that they contain images from a forward-pointing camera and have camera calibration data available. Additionally, it is beneficial if the dataset has lidar scans or equivalent available, as these are often required by supervised methods and highly beneficial for SSL-based methods as it unlocks the possibility of doing online evaluation during training.

2.7.0.1 KITTI

The KITTI Vision Benchmark Suite [33] is a well-known benchmark for multiple tasks in autonomous driving. This dataset is collected with a single platform equipped with a 360 Velodyne lidar, a stereo camera rig, and a GPS. The data is collected in the Karlsruhe region in Germany and is fully annotated with bounding boxes, semantic segmentation, and odometry ground truth. Calibration data for all the sensors are also available. After being available since 2012, this dataset has risen to become the "de-facto" benchmark for a lot of autonomous driving tasks, with dense depth estimation being one of them.

For a dense depth estimation task like this thesis explores approximately 47 000 frames with camera and lidar data available. Although the lidar data is not necessary for an SSL-based method, the lidar data can be utilized to do an online evaluation of the predicted depth values. However, most SSL-based methods use a subset consisting of approximately 40 000 frames selected by Zhou *et al.* [44], which again is a subset from the subset created by Eigen *et al.* [37] of KITTI. These subsets are commonly used in monocular dense depth estimation tasks due to the frames consisting of little to no motion have been removed.

2.7.0.2 Cityscapes

The Cityscapes benchmark suite [48] is a dataset that focuses on the semantic understanding of autonomous driving-related scenes. This dataset was put together in 2016 in a collaboration between Daimler, TU Darmstadt, MPI Informatics, and TU Dresden. The data is collected from 50 urban cities, primarily in Germany, consisting of 5000 annotated high-quality, dense pixel annotations and 20 000 more coarse, polygonally-based annotations. Even though there only exist 25 000 labeled frames, around 40 000 extra unlabeled frames can be utilized for this use case, as the annotations are not needed for this use case. The total number of available frames is around 77 000.

2.7.0.3 Lyft

Level5 is a Lyft-based company that develops autonomous driving solutions for the Lyft network. In 2020, they presented their "Level 5 open data" dataset [49] for their Motion Prediction competition. The dataset consists of a prediction dataset that contains motion data on multiple types of traffic agents and a perception dataset containing raw sensor data from a sensor suite consisting of cameras and lidars. Only counting the forward-pointing camera, the dataset offers a total of 17 000 usable frames, all accompanied by a high-resolution lidar scan.

2.7.0.4 DDAD

The "Dense Depth for Autonomous Driving" dataset was released by the Toyota Research Institute (TRI) together with their *PackNet* paper in 2020 [30]. The dataset is collected by Toyota's autonomous fleet located in San Francisco, Tokyo, Cambridge, and Detroit. It focuses on long-range dense depth detection and contains accurate 360 lidar scans for up to 250 meters, accompanied by high-resolution cameras covering a 360 view around the capturing vehicle. Currently, the dataset serves as a benchmark for long-range dense depth estimation tasks and the primary dataset for TRI's DDAD depth challenge.

2.7.0.5 Waymo

The Waymo Open Dataset [50] was first released to the public in August 2019 as a part of their set of challenges for different tasks in autonomous driving, consisting mainly of 2D and 3D object detection. This dataset has later been updated with more data and annotations, at the latest in March 2021, where they included a new motion dataset. Currently, the dataset consists of over 50 000 usable frames captured in and around San Francisco, California. Lidar scans accompany all the available frames. These can be used to generate spare depth maps.

Chapter 3

Methodology

This chapter discusses which dense depth estimation methods have been selected and why and which datasets will be used to train these methods. Following is an in-depth explanation of selected methods, focusing on design choices and inner workings. The subsequent section covers how the selected dataset was extracted and how a shared data organization setup was developed. Some changes and tweaks needed to be done to the selected methods due to the selected methods not natively supporting the selected datasets. These modifications are explained in the penultimate section, followed by some details regarding evaluation metrics and computing hardware used during training.

3.1 Choice of Methods

Section 2.6 discussed some of the historically significant improvements and some of the current SotA models. This section will discuss which SSL-based methods are candidates to be trained and which supervised method will be used. Following is also a discussion of which datasets will be used to train the selected methods. The section concludes with which methods and datasets will be selected to be used in this thesis.

3.1.1 Candidates - Self-supervised-based Methods

3.1.1.1 Monodepth2

Monodepth2 [29] was the primary method used in the specialization project *Self supervised Learning - Project Thesis* by Galteland [51] previous to this thesis. This method showed great potentials on some of the datasets used there. However, in specialization project did not include any evaluation of the actual depth data generated by the method. Using Monodepth2 in this thesis makes for a great opportunity to expand upon the results found in the project thesis by doing an actual evaluation of the results by utilizing lidar scans. On the KITTI benchmark dataset, Monodepth2 achieves an absolute relative error of 0.109 when using monocular data. The absolute relative error metric is described in Section 3.8.2.

3.1.1.2 PackNet

PackNet [30] shows some exciting potential with its ability to compress and decompress the image with close to lossless compression while still utilizing the power of the convolutional operation. The fact that it also needs no pre-trained weights is a bonus as well. Some of the candidate datasets also have velocity data available, unlocking PackNet's ability to become semi-supervised. PackNet is, all in all, a tempting candidate to be used in this thesis. PackNet achieves an absolute relative error of 0.107 on KITTI when using monocular data.

3.1.1.3 Feature Depth

Feature-metric loss for Self-supervised Learning of Depth and Egomotion by Shu *et al.* [46] shows that simply improving the architecture with a separate loss function focusing on improving the problems with the photometric error-based loss functions can give a considerable improvement and advantage over other SSL-based dense depth detection networks. Exploring how this behavior extends itself to other datasets and the fact that it is the current overall SotA on KITTI is both valid reasons to continue with Feature Depth. On the KITTI, Feature Depth is the current SotA unsupervised method when using both monocular and stereo input, and the runner up when only using monocular data, having an absolute relative error of 0.104 for the monocular only category.

3.1.1.4 Manydepth

While being released a couple of months after starting writing this thesis, Manydepth [47] could not go unrecognized. While not being the first SSL-based dense depth system to utilize multiple frames at test time, Manydepth is the first one to do it using cost-volumes, and in many ways, revolutionized the field of SSL-based dense depth estimation with this novel implementation. Manydepth is in many ways what Monodepth [28], and Monodepth2 [29] were in their time; an inspiration and sandbox for novel ideas and techniques. Therefore, Manydepth is a strong candidate to be used further on in this thesis. Manydepth is the current SotA for monocular only sequences, with its absolute relative error of 0.087. This score is achieved when using ResNet50 instead of ResNet18 for the depth model.

3.1.2 Candidates - Supervised Methods

This section contains the different candidates for the supervised methods.

3.1.2.1 Ada Bins

AdaBins [41] serves as one of the SotA methods for all dense depth estimation tasks. With its semi transformer-based architecture, it is an attractive choice to use in this thesis, especially since it achieves the best results out of all other methods on the KITTI dataset, which is highly relevant for this use case. AdaBins is the current SotA on KITTI, achieving an impressive 0.058 absolute relative error.

3.1.2.2 DPT

The only other candidate that can compete with AdaBins when looking at innovation and performance is the recently released *Vision Transformers for Dense Prediction* paper by Ranftl *et al.* [42] This method is the first to utilize the vision transformer by Dosovitskiy *et al.* [17] in a dense prediction task. It also achieves SotA results on some datasets. Another exciting advantage is that the DPT architecture can function as the backend for any dense prediction task, meaning that it can also easily be applied to do, e.g., semantic segmentation. DPT achieves an absolute relative error of 0.062.

3.1.3 Candidates - Datasets

As mentioned, this thesis intended to use data from the NAP-lab vehicle and focus on how these methods would function in a more arctic environment. However, due to complications with both getting access to the data and NAP-lab recently acquired a new vehicle, this data only became available a week before the thesis's original due date. Therefore, other more acknowledged datasets had to be considered, as relying on using data from NAP-lab was out of the question. Thus, more acknowledged datasets had to be considered a replacement.

It was decided that the methods selected should be trained with a benchmarking dataset so that the reported results of the selected methods could be verified. In addition, two other datasets should be used. It is preferable is these datasets do not have any significant publications documenting using the dataset for dense depth estimation tasks. Therefore, a benchmarking dataset and two other datasets needed to be chosen from the documented datasets in Section 2.7. As NAP-lab became available only a week before the due date of this thesis, training with NAP-lab data would only be performed as an addition to the benchmarking and two other datasets.

3.1.3.1 Benchmarking Dataset

As both KITTI [33] and Cityscapes [48] are well-known datasets used for benchmarking, this thesis wanted to use one of these to confirm that the results reported in the papers about the selected methods were reliable. KITTI was chosen as all the candidate methods have reported results for KITTI.

3.1.3.2 Training Datasets

The Waymo dataset is a reasonably new dataset containing many high-quality frames with accompanying lidar scans. However, due to the problems that were experienced in the specialization project [51] when using the dataset to train Monodepth2 [29], it was decided that this dataset would not be used again due to the problem still being unexplainable, even by Goudard *et al.* [1]

Another candidate is the Lyft Level 5 dataset [49]. While being on the smaller side regarding available frames, [51] still showed positive results when using this dataset on Monodepth2 [29]. Though in retrospect, it was discovered that the dataset contained images in two different

[1] Link to conversation with Goudard in a GitHub issue can be found here

dimensions, where approximately 5 000 frames are of size 1920×1080, and the other 17 000 frames are of size 1224×1024. While this is not a problem for the possibility to train methods themselves, due to the images being resized in the preprocessing steps, it can affect the final accuracy of the resulting models. [51] did not consider this, and the results reported here might not be as reliable nor accurate as it has the potential to be. Therefore, the Lyft level 5 dataset is a great candidate to be used in this thesis so that the results from [51] can be verified.

The final candidate is the "Dense Depth for Autonomous Driving" (DDAD) [30] by the Toyota Research Institute. There are currently no other reported cases of using the selected methods with this dataset, as per the writing of this thesis. Hence, DDAD is an excellent candidate to use in this thesis.

3.1.3.3 Using Syntesized Data

At one point, it was discussed whether or not to use synthesized data from a simulator like CARLA [52], AirSim [53], or utilizing GAN-based methods to make synthesized data look more realistic [54]. It is tempting to be able to generate potentially an unlimited amount of frames with perfect depth information. However, it was concluded that this thesis should focus on SSL in itself and that branching out by looking into synthesized data could lose the running theme of the thesis. Looking into a combination of using SSL-based methods for dense depth detection with synthesized data by itself or in combination with real data could be a potential future thesis.

3.1.4 Conclusion

After having looked at the different candidates, the candidates were assessed by their different contributions. This section contains a summary of which candidates were chosen with an explanation of why.

3.1.4.1 Self-supervised Methods

Monodepth2 was the primary method used in the specialization project. Therefore, it has already been relatively explored and explained. As mentioned in the candidate description, it would have been interesting to evaluate Monodepth2's performance on the Lyft level 5 dataset with the evaluation metrics introduced in Section 3.8.2. However, dedicating this thesis to look into an already explored method would lead to a lot of repeated work and conclusions. Therefore, Monodepth2 is dropped.

The significant improvement in performance that the feature-metric loss showed in the Feature Depth method can not go unnoticed. Therefore, this method is one of the selected methods.

PackNet was one of the chosen methods at an earlier point in the thesis. However, as Manydepth was released in late March, it had to be dropped to explore Manydepth in greater detail, as Manydepth introduces such a novel idea and will likely work as a breeding ground for many other novel ideas in the field of SSL-based dense depth estimation.

3.1.4.2 Supervised Method

Only one of the two presented candidates could be selected as the supervised method for this thesis. AdaBins is a tempting choice as it is the best performing method on the KITTI dataset. Due to the available frames in KITTI being reasonably low in the view of a transformer, which usually requires vast amounts of data to perform well, the DPT method probably does not have sufficient data to perform at its best, even if it is pretrained with other datasets. This means that the DPT method possibly still has more potential to improve. The deciding factor between AdaBins and DPT is that DPT can generalize to any dense estimation task, which would also be interesting to investigate as an addition to the other experiments. Therefore, this thesis will use the DPT as its supervised method.

3.1.4.3 Dataset

KITTI was early on selected to function as the benchmarking and validation-of-results dataset for this thesis. As mentioned, the specialization project related to this thesis by Galteland [51] experienced some mysterious problems with the Waymo dataset. Some possible answers to why were discussed in the specialization project, but none of the suggested fixes were ever tested. It is tempting to use this thesis as a validation of the suggested fixes, but due to not even Goudard having any thoughts on why this happened, choosing Waymo felt too risky if none of the solutions would have worked, even if it would have worked as an excellent way to confirm the suspicions in the specialization project. Therefore, Waymo is not explored any further in this thesis.

The specialization project showed that the Lyft dataset had great potential in a dense depth estimation use case. As discussed earlier, it was discovered that the training split used in the specialization project contained different-sized images, which most likely affected the results. Consequently, the Lyft level 5 dataset will continue to be used to confirm the results from 51 and verify that using only the same-sized images indeed affects the results.

To not ignore the contributions of PackNet [30], the thesis will also continue with the DDAD dataset as it is reasonably new and has little to no other published results other than PackNet itself regarding performance on a dense depth data estimation task. Therefore, it is interesting to validate that the results PackNet showed using the dataset can be transferred to other dense depth estimation methods and to see if any other methods can beat PackNet itself on the dataset, as PackNet reported better results for the PackNet architecture. However, other benchmarks like KITTI show that Monodepth2 performs better than the SSL-PackNet architecture [2]

3.1.4.4 NAP-lab Data

A set of videos from the NAP-lab vehicles became available at a late stage in the thesis. It was decided that the selected Manydepth method should be trained with NAP-lab data, as sufficient infrastructure was already created when modifying the method to work with the Lyft and DDAD datasets. No lidar scans would be used due to time constraints, meaning that it

[2] Based on the leaderboard on paperswithcode.com

would not be possible to evaluate the actual depth data. Nevertheless, looking at the generated depth images would indicate the expected performance.

3.2 Self-supervised Learning-based Dense Depth Estimation

As mentioned in Section 2.6, Solutions using SSL are primarily based on using separate models θ_{depth} and θ_{pose} for predicting the depth D_t and relative pose $T_{t\to s}$ between two frames I_t and I_s. The selected methods of Manydepth [47] and Feature Depth [46] both utilize these ideas. Thus, this section will cover the basics of the geometry models and introduce other needed knowledge.

3.2.1 Camera Intrinsic

A camera intrinsic matrix K [55] is used to transform a 3D point into a 2D pixel. The matrix describes a perspective projection performed with an ideal pinhole camera. It is defined as:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

The values f_x and f_y represents values for the focal length of the camera used specified in pixels. f_x is defined as $f_x = \frac{f}{m_x}$ and f_y as $f_y = \frac{f}{m_y}$, where m_x and m_y represents the real-life size of a pixel, and f is the actual focal length specified as distance. c_x and c_y represent the principal points, which is the ideal center of the image. The intrinsic matrix K matrix is usually found by calibrating the camera, often performed with a checkerboard [56].

3.2.2 How Self-supervised Methods Learn to predict Depth

As mentioned in Section 2.4, a Self-supervised learning problem uses no external ground truth values. However, SSL tasks have shown that one does not need information about the ground truth to be able to learn something about the core problem itself. For some tasks, one can even solve a different problem and use the intermediate info from solving that problem to solve the actual problem at hand. SSL-based methods for dense depth detection does precisely this. Instead of solving the ill-posed problem of regressing a depth value for all pixels in an image, with no supervision signal available, these methods solve this by re-imagining the core problem itself.

By first describing the problem in a more general manner, this section will explain how SSL-based dense depth estimation methods utilize geometric view synthesis as the core learning problem to learn networks to predict depth.

3.2.2.1 Problem Description

It is best to take a step back to get a better understanding of these problems. Imagine having a point cloud consisting of points in 3D space. When viewing this point cloud on a screen, one is watching a 2D projection of this 3D data structure from an imagined camera at a position in a

3D space. If one wants to view another 2D projection of the point cloud, one can transform the camera to a new position in 3D space, with a transform that can be in all 6 degrees of freedom. After having translated the camera, one is left with a new 2D projection which changed how the image viewed on a screen looks, but the 3D point cloud is still unchanged and is constant for a scene no matter where one chooses to place the camera.

Another key realization here is that the 2D projection precisely represents the data in 3D space to 2D space. However, if one knows how the points were projected, one can not necessarily project these back into 3D, as one projects to 2D, the depth coordinate is lost, often the Z coordinate. Thus, the depth value Z is needed to project back into 3D space.

3.2.2.2 Geometric View Synthesis

Suppose one has a source image I_s and target image I_t, which captures a scene from two different viewpoints relatively close to each other in the world. The relative difference between these two viewpoints is a pose $T_{s\to t}$, which indicates a six degrees of freedom transformation matrix between the source and target viewpoint, which essentially is a transformation in any direction and with any rotation in 3D space. This relative position is unknown, and so are the depth values for the pixels in the two viewpoints 2D. If one were to have both the depth values and the relative position between the images, one could use geometric view synthesis to reconstruct the target image only from the source image.

Geometric view synthesis starts by first converting the pixels in the source image I_t to the camera frame using the inverse $proj$. This can be done as follows:

$$proj(D_t, p)^{-1} = D_t \left(\frac{x - c_x}{f_x}, \frac{y - c_y}{f_y}, 1 \right)^\top \tag{3.2}$$

The inverse of the $proj$ function transforms a 2D pixel p to a 3D point P by using the pixel p together with predicted depth D_t. The f_x, f_y, c_x and c_y parameters are from the associated intrinsic matrix K.

The pose $T_{s\to t}$ describes the relative pose between the source and target in the camera frame. However, the pose between the target in the camera frame and the pixel frame source is needed to project into the pixel frame. This transform is achieved by multiplying the relative pose $T_{s\to t}$ the intrinsic matrix K:

$$T_s^C = T_{s\to t} \cdot K \tag{3.3}$$

With this one can project the points into the target image's pixel frame:

$$\omega\left(D_t, T_s^C, p\right) = T_s^C \cdot proj^{-1}(D_t, p) \tag{3.4}$$

ω produces a 2D coordinate frame, where the value at index (i, j) is a 2D coordinate representing the same value in the source frame. These 2D coordinates are float values and might represent a coordinate that lays between multiple pixels. To create the final synthesized image, one uses a bilinear sampler to sample pixels from the source image:

$$I_{s\to t} = I_s\Big\langle \omega\left(D_t, T_s^C, p\right)\Big\rangle \tag{3.5}$$

Here, $I_{s\to t}$ is the synthesized version of I_t. $\Big\langle\Big\rangle$ denotes the bilinear sampler.

To summarize; this function inversely projects a 2D pixels p in the source image I_s into the camera frame in 3D space, into 2D coordinates in the pixel frame of I_t, where a pixel at index (x, y) represents a pixel (i, j) in the source image I_s. These coordinates are sampled from the source image I_s, and one is left with a synthesized version of I_t, only created with pixels from I_s.

3.2.2.3 Learning Depth

The approach mentioned above has a fundamental problem. This approach assumes that the depth D_t and relative pose $T_{s\to t}$ are known and ideally perfectly correct. With perfect knowledge of both the depth and the pose, one can almost perfectly synthesize I_t. However, if either the depth or pose is wrong, the resulting synthesized image will also be incorrect, leading to a synthesized image that is not similar to the target image.

This realization is the core of self-supervised dense depth estimation. By having a model θ_{depth} that predicts a dense depth map D_t and a different model θ_{pose} that predicts a six degrees of freedom pose between two images, one can use the geometric view synthesis as the core learning problem. By introducing functions that measure the difference between two images, one can use the fact that to construct an image $I_{s\to t}$ from I_s, the depth D_t and pose $T_{s\to t}$ needs to be correct. The better depth and pose predicted, the closer the image $I_{s\to t}$ will look to the original I_t. By jointly training both models and using loss functions that are based on measuring the similarity between a target image and its synthesized counterpart, the models θ_{depth} and θ_{pose} are thus forced to become good at predicting depth D_t and relative poses $T_{s\to t}$ for the loss to decrease and eventually converge. In other words, the only way to do it consistently is to be correct.

3.2.2.4 Requirements

This approach does have a few requirements. The first requirement is that some motion between the source and target frames is needed. Without it, one would not have any significant changes in the image, and no disparities could be created. On the other hand, this motion cannot be too big either, leaving it hard or even impossible to reconstruct the target image from the source image. The final requirement is that the captured scene is static. This requirement is a bit hard to fulfill for datasets from autonomous vehicles, as they operate in a domain where other vehicles are moving around, resulting in moving objects in the scene. If this requirement is unfulfilled, one will end up with a "punching hole" effect, where the model will predict an infinite depth for the moving object. This requirement applies mainly for objects moving in the same direction as the ego-vehicle, with a velocity close to the ego-vehicle itself. Objects moving perpendicularly or oppositely to the ego-vehicle often do not have a severe reaction compared to other moving objects. This is because they can be interpreted as static by the models and that their change in position in the scene is caused by ego-motion.

3.2.3 Photometric Error

Some steps are needed to measure the difference between two images correctly. The first and most obvious way of measuring the difference between two images is to take the L1 distance or the absolute difference between two images:

$$L1(I_a, I_b) = \|I_a - I_b\|_1 \tag{3.6}$$

Although, the L1 difference only measures the differences in pixels located at the same position in the I_a and I_b. However, these differences do not always represent differences in other aspects of an image. Therefore, most photometric errors use structural similarities (SSIM) [57], which measures the difference between luminosity and structure on a pixel level. A perfect copy of I_a would give an SSIM of 1, while a perfect imperfection would result in an SSIM of -1. With these two error measures, one can create a combined, weighed photometric error that utilizes both SSIM and L1 distance:

$$pe(I_a, I_b) = \alpha \frac{1 - \text{SSIM}(I_a, I_b)}{2} + (1 - \alpha)\|I_a - I_b\|_1 \tag{3.7}$$

Virtually all SotA self-supervised papers use this photometric error as a baseline for their photometric error, and they all use α set to 0.85.

3.2.4 Basic Architecture

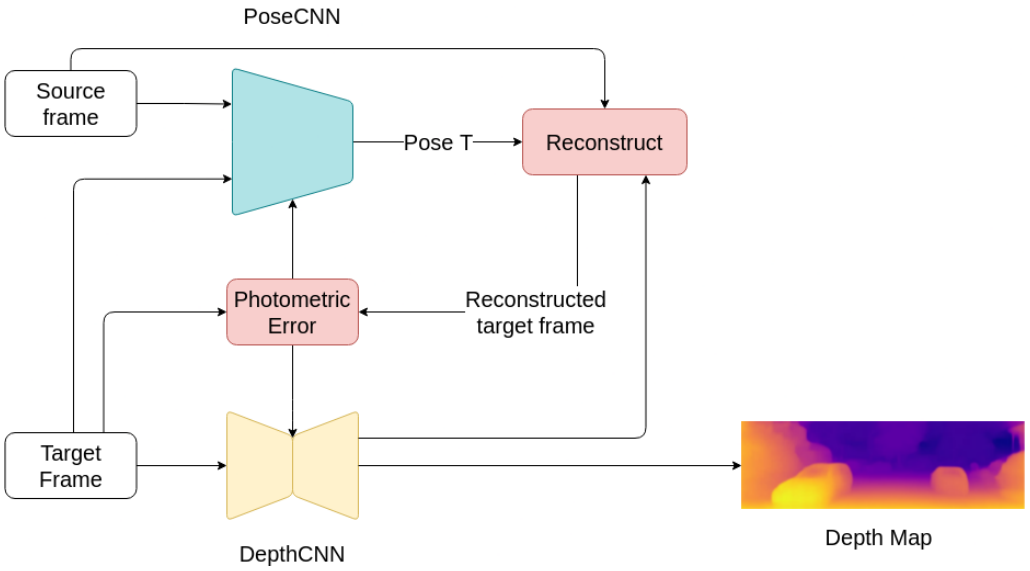


Figure 3.1: Illustration of a basic self-supervised depth estimation architecture.

A basic architecture for a self-supervised depth estimator is shown in Figure 3.1. A target and source frame are fed into a pose model θ_{pose} that outputs the pose $T_{s \to t}$. This pose is used together with the hallucinated depth D_t generated by a depth model θ_{depth} for the frame I_t to reconstruct the target frame from the source frame's pixels. These are used to calculate the photometric error between the images, which is used as the loss for both the pose and depth

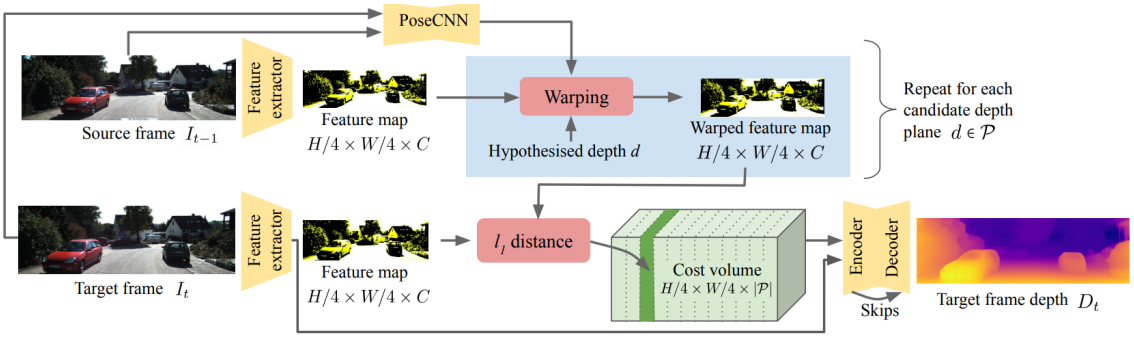
models. The only way for the photometric loss to improve is by predicting the correct depth and pose for the source and target images. Thus, the depth model will generate correct depth values.

3.3 Manydepth

This section will go into the details of how Manydepth by Watson *et al.* [47] works. The section starts by giving a general overview of the architecture of Manydepth, followed by an explanation of the different losses used. The rest of the section will focus on the novel way of using sequences of images rather than using a single image at inference using a cost-volume, and the novel consistency loss to encourage the model to ignore the cost volume when it fails.

3.3.1 Architecture

The Manydepth system consists of three separate parts. The first part is a depth architecture that uses multiple frames I, by using a cost-volume as an additional input to an encoder-decoder network that outputs a depth frame D. The second part is a pose network that can predict the relative movement between two frames I_t and I_{t+n}. Finally, the last part is an encoder-decoder consistency network, similar to the network used in the depth architecture but without the modifications needed to use cost-volumes, making it identical to the depth network from Monodepth2.



(a) Depth architecture

(b) Pose Network

(c) Consistency Network

Figure 3.2: Illustration of Manydepth's architecture. The depth architecture illustration is adapted from Watson *et al.* [47], while the pose and consistency network illustrations is adapted from Goudard *et al.* [29]

3.3.1.1 Depth Architecture

The overall depth architecture θ_{depth} is substantially changed from Monodepth2, which consists primarily of a single encoder-decoder network. However, for Manydepth, the depth architecture consists of three components, with the first being the feature extractor. This feature extractor uses the first five ResNet18 layers [10] to turn images into features. These features are fed into a cost volume, which is the second component in the model and discussed further in Section 3.3.3. The image features and the cost volume are then fed into the remaining convolutional layers of ResNet18, which completes the encoder part of the depth architecture.

The decoder is the final of the three main parts of the θ_{depth} model. This decoder follows Monodepth2's decoder. It primarily consists of 'upconv' layers, which use upsampling in the form of nearest-neighbor upsampling and a convolutional layer to bring the features from the decoder back into an image representation. The final output is a depth image D.

3.3.1.2 Pose Network

The pose network θ_{pose} consists of a ResNet18 network modified to accept two frames, or six channels, instead of one. The output from the network is a single six degrees of freedom relative pose between the two input frames.

3.3.1.3 Consistency Network

The consistency network $\theta_{consistency}$ is identical to the depth network from Monodepth [29]. It consists of a ResNet18 encoder and a decoder similar to θ_{depth}'s decoder.

3.3.2 Reprojection-based Training

Manydepth is based on self-supervised reprojection based training, described in Section 3.2.2. This training uses an frame I_t, a frame at time t together with frames that are temporally close $\{I_{t-n}, I_{t+n}\}$. Together with the estimated depth D_t from θ_{depth} and the pose $T_{t\rightarrow t+n}$ from θ_{pose}, one can synthesize the frame I_t from different viewpoints using temporally close frames. In Watson *et al.* experiments, the temporal frames are $\{I_{t+n}, n \in \{-1, +1\}$. With this, one can learn synthesize I_t from only using pixels from $\{I_{t-1}, I_{t-1}\}$. This synthesizing is done by using Equation (3.5), where the result is a synthesized frame $I_{t+n\rightarrow t}$ Here, $I_{t+n\rightarrow t}$ denotes synthesizing I_t from I_{t+n}, where $n \in \{-1, +1\}$.

Equation (3.5) uses the intrinsic matrix K_t. The intrinsic matrix K_t is assumed to be identical for all temporally close frames, but it is worth mentioning that this is not a requirement. Due to most datasets being collected using a fleet of autonomous vehicles, all with slight changes in their camera setups, it is not uncommon for the intrinsic matrix K_t to differ between the collected scenes. Therefore, the intrinsic matrix K_t is set, during training, to be the intrinsic matrix K_t for the vehicle that collected the data.

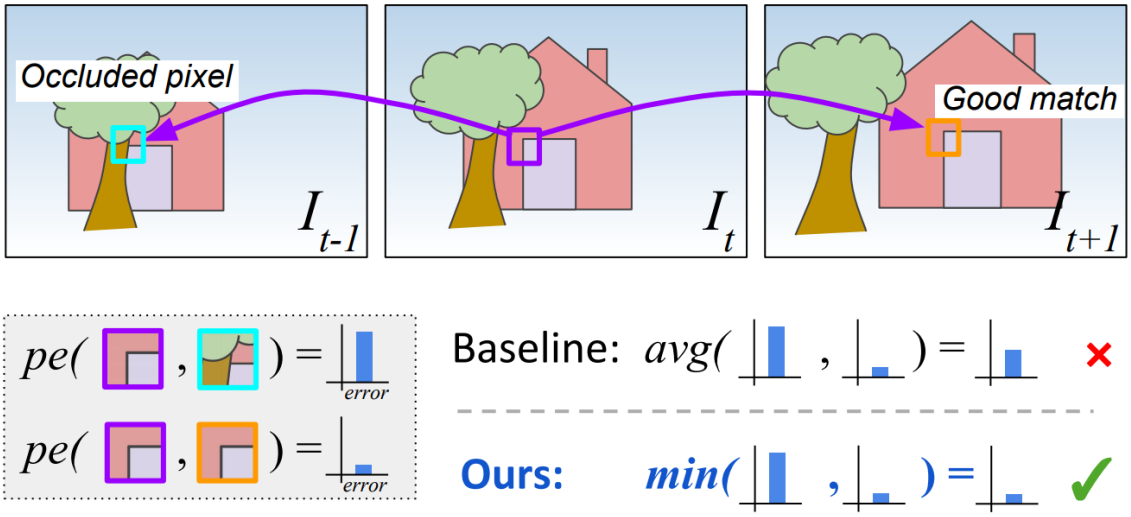


Figure 3.3: Example of using minimum appearance loss. Using the minimum photometric error can prevent false positive cases of high photometric error. Adapted from Goudard *et al.* [29]

Similarly to Monodepth2 [29], Manydepth optimizes the loss for the best matching source image. This loss is based on a photometric error from Equation (3.7).
Monodepth2 commented that the use of average pe, which many other SSL solutions use, can cause problems for pixels that are either out-of-frame or hidden in the target image I_t but exist in the temporally close frames. These cases result in an artificially high pe for that region. An example is shown in If using the average pe, the resulting pe would be a lot higher than if using the minimum. Thus, Manydepth follows the conclusion made by Monodepth2 and uses the minimum, rather than the average pe as their reconstruction loss:

$$L_p = \min_n pe(I_t, I_{t+n \to t}) \tag{3.8}$$

This loss is calculated over for the frames at four different scales of the image. Manydepth uses frames of size $\{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$ times the original frames' size.

In addition to using the reprojection loss, Manydepth also adapts the edge-aware smoothness loss from the works of Goudard *et al.* [28, 29, 58], which encourages the depths to be locally smooth. This encouragement is done by enforcing a $L1$ penalty on the disparity gradients ∂d:

$$L_{smooth} = |\partial_x d_t^*| e^{-|\partial_x I_t|} + \|\partial_y d_t^*| e^{-|\partial_y I_t|} \tag{3.9}$$

Where $d_t^* = d_t / \overline{d_t}$ is the mean normalized inverse depth used to discourage shrinkage of the estimated depths.

The main reason to encourage locally smooth depths is that an object in a scene has depth values close to each other. Taking, for instance, a car in an image, its minimum, and maximum associated depth value will be at most equal to the length of the car. If a model tried to associate randomly spots of depth values that result in a max difference that surpasses the length of the

car, one would assume that these associated depth values are wrong. Therefore, the depth values are encouraged to be locally smooth.

3.3.3 Cost Volume

This cost volume represents the geometric compatibility at different depths in the target frame I_t and the temporally close frames. A set of planes $\mathcal{P}$ is generated and added perpendicularly to the optical axis of I_t. These planes are linearly spaced between the minimum depth d_{min} and the maximum depth d_{max}. Each of these frames is encoded as a feature map F_t using a feature extractor. These feature maps are then warped to the viewpoint of I_t using all of the available depths $d \in \mathcal{P}$ using the same idea as Equation (3.5), creating a warped feature map $F_{t+n \to t,d}$.

With this, one can create a cost volume consisting of the difference between the warped feature map and the feature map from the target frame I_t for all depth $d \in \mathcal{P}$, taking the average for all the temporally close source frames. One ends up with is a $H \times W \times |\mathcal{P}|$ structure that effectively holds the information about the likelihood for an arbitrary pixel $\{x, y\}$ to be correctly classified as depth d for all $d \in \mathcal{P}$. After being created, this cost volume is concatenated with the feature maps F_t and sent to the convolutional decoder that predicts the depth D_t.

3.3.3.1 Handling Special Cases

Due to the usage of past frames not being a requirement, the model should handle cases where only a single frame I_t is used at test time. Thus, the cost volume is replaced with an empty zero tensor with a probability p during training. This replacement is done to encourage the model to not rely solely on the information in the past frames.

Another problematic case when using multiple frames is when there is no movement between I_t and $I_n, -N \leq n \leq N$. To combat this, I_n is replaced with an augmented version of I_t, but with using I_n in L_p from Equation (3.8). This encourages the network to predict valid depths when the cost volume is based on images with no change in pose.

3.3.3.2 Adaptive Cost Volumes

Manydepth proposes the novel idea of using adaptive cost volumes to better predict depths for an arbitrary values of d_{min} and d_{max}. In a standard cost volume, the values d_{min} and d_{max} needs to be specified before training. However, in an SSL setting like this, these values are not available. Only data from the images themselves are used, and one can not depend on having these values available.

Manydepth solves this by learning d_{min} and d_{max} from the data itself. These parameters are learned using the predicted D_t, where d_{min} and d_{max} are computed as the average min and max of each D_t in a training batch. These values are then updated and stored after each batch. After a set number of epochs, these values are frozen and not updated anymore. This freezing also applies for the weights of the $\theta_{consistency}$.

3.3.4 Consistency Loss

The use of these adaptive cost volumes alone does not yield any significant improvements. In fact, using this idea with reprojection-based training yields far worse results than not using the cost volume at all, with punching-hole-like artifacts for moving objects as one of the most significant artifacts. These holes occur because self-supervised monocular training operates under the assumption of a static scene and a moving camera. When these assumptions are made, e.g., for moving objects like cars or pedestrians, these holes can occur. [29] attempted to fix this with an automatic masking method that removed the pixels that were suspected of containing these kinds of objects. This failure is because the cost volume can produce unreliable results for moving objects, like moving objects and low-textured areas, e.g., poorly lit surfaces.

When training with both the reprojection-based systems and the cost volume, the total system can become overly dependent on the cost volume, which will generate wrong predictions for the vulnerable areas. Therefore, the network needs to learn when not to trust the data from the cost volume to improve this behavior. This problem could be partially solved by, e.g., using a separate semantic segmentation network to identify moving objects [59] or automatically mask the stationary pixels between the temporally close frames [29].

Manydepth solves this by training a single-image-based depth network $\theta_{consistency}$ simultaneously as training the multi-image-based network with cost volumes, as the single-image-based does have the problems that a cost-volume supported approach introduces. It is worth mentioning that using data from the single-image-based network does not solve the problem itself, as the hole artifacts also are apparent here. However, the artifacts are not as apparent for moving objects as when using cost-volumes.

As the cost-volume-based approach generally performs better than a single-image-based approach, it is essential to identify which pixels are affected by the cost-volume weaknesses so that only the affected areas are replaced. The single-image-based network $\theta_{consistency}$ outputs a depth map $\hat{D}_t$ of the same dimensions as D_t for all training images. To identify which pixels are reliable, Manydepth concludes that the cost volume can be counted as reliable when the depth $\hat{D}_t$ is close to the `argmin` of the cost volume, also denoted as D_{cv} as the depth from the cost volume. The `argmin` of the cost volume translates to the lowest $L1$ distance between the warped feature map generated by warping the source frame I_{t-1} and the feature map from I_t. With this, a binary mask is defined as:

$$M = \max\left(\frac{D_{cv} - \hat{D}_t}{\hat{D}_t}, \frac{\hat{D}_t - D_{cv}}{D_{cv}}\right) > 1 \tag{3.10}$$

Which translates to the binary mask being 1 in regions where $\hat{D}_t$ and D_{cv} have a significant difference and denotes the areas of unreliable pixels from the multi-image model $\theta_{consistency}$. With these unreliable pixels known, one can define a new loss value $L_{consistency}$ as:

$$L_{consistency} = \sum M|D_t - \hat{D}_t| \tag{3.11}$$

This loss value will encourage predictions D_t from θ_{depth} to be similar to predictions $\hat{D}_t$ from $\theta_{consistency}$ for unreliable pixels defined by M.

The final loss value is a combination of all the previously discussed loss values:

$$L = (1-M)L_p + L_{consistency} + L_{smooth} \tag{3.12}$$

The $(1-M)$ term before L_p results in only calculating L_p for reliable pixels.

3.4 Feature Depth

The main contribution from *Feature-metric Loss for Self-supervised Learning of Depth and Egomotion* by Shu *et al.* [46] is their feature-metric loss. This section will explain how Shu *et al.* argues for why the feature-metric loss improves dense depth prediction and how their "Feature Depth" architecture is constructed.

3.4.1 Architecture

In this paper, Shu *et al.* [46] takes inspiration from Monodepth2 [29] for its novel Feature Depth architecture. Here, both the depth network θ_{depth} and the pose network θ_{pose} utilizes the ResNet [10] architecture, much like Monodepth2 and Manydepth [47]. However, this architecture chooses to use ResNet50 for its depth network rather than using ResNet18 as Monodepth2. The pose network is modified to use two input frames, just as Monodepth2. These two networks are essentially identical to the θ_{pose} and $\theta_{consistency}$, as the Feature Depth architecture only uses a single input image, and thus no cost volumes. These networks are introduced and explained in Section 3.3.1.

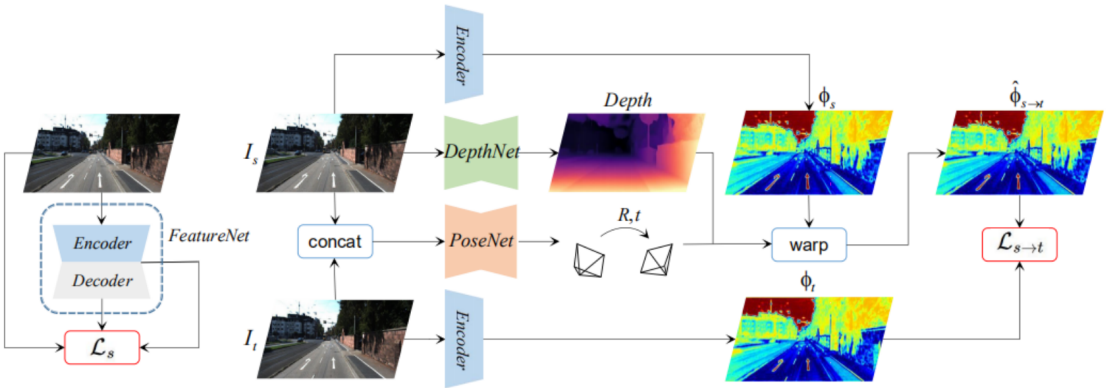


Figure 3.4: Illustration of Feature Depth's final architecture. It is worth noting that the encoders used after I_s and I_t is the encoder from the FeatureNet. It is also worth noting that even though the losses L_s and $L_{s\to t}$ is separated, they both count toward the final loss. Adapted from Shu *et al.* [46]

3.4.1.1 FeatureNet

FeatureNet is one of the novel contributions from Shu *et al.*. This network is responsible for creating features ϕ that can help the photometric error in areas where it suffers, e.g., dark and low-textured areas. This network consists of an encoder-decoder structure, where only

the encoder part of the network is used together with the other parts of the architecture. The FeatureNet utilizes a ResNet50 [10] architecture, with the final fully-connected layers are removed as its encoder. The decoder consists of five "upconv" layers, each consisting of a 3×3 convolutional layer followed by a bilinear upsampling layer.

The network can be trained either jointly together with the θ_{depth} and θ_{pose}, or separately with pretrained θ_{depth} and θ_{pose} models.

3.4.2 Reconstruction Loss

The reconstruction loss is the loss used when finding the difference between the target and source images. Feature Depth uses both a single and multi-view reconstruction loss.

3.4.2.1 Multi-view Reconstruction Loss

Feature Depth follows Monodepth2, thus also Manydepth in its multi-view reconstruction loss, defining its loss as:

$$L_{s \to t} = \sum_{p} pe\left(I_s(\hat{p}), I_t(p)\right) \tag{3.13}$$

Here, pe is the photometric error defined in Equation (3.7). $\hat{p}$ is the pixels p from the synthesized image $I_{s \to t}$, and p is pixels from the target image I_t.

However, Feature Depth comments on the use of photometric loss by itself as being fundamentally problematic. The problem with photometric loss is that a low photometric loss does not necessarily indicate a correctly predicted depth or pose. This behavior is especially apparent in close to textureless areas, e.g., poorly lit areas or buildings. Feature Depth shows that the problem can be formally described from the optimization perspective by deriving the gradients for the depth and egomotion:

$$\frac{\partial L_{s \to t}}{\partial D_t} = \frac{\partial pe\left(I_s(\hat{p}), I_t(p)\right)}{\partial I_s(\hat{p})} \cdot \frac{\partial I_s(\hat{p})}{\partial \hat{p}} \cdot \frac{\partial \hat{p}}{\partial D_t} \tag{3.14}$$

$$\frac{\partial L_{s \to t}}{\partial T_{s \to t}} = \sum_{p} \frac{\partial pe\left(I_s(\hat{p}), I_t(p)\right)}{\partial I_s(\hat{p})} \cdot \frac{\partial I_s(\hat{p})}{\partial \hat{p}} \cdot \frac{\partial \hat{p}}{\partial T_{s \to t}} \tag{3.15}$$

The gradients $\frac{\partial I_s}{\partial \hat{p}}$ are the gradients for the image. In the textureless regions, these gradients are close to zero in both cases. Due to these problems, Feature Depth proposes to use features $\phi_s(p)$ found in the source image instead of using the actual image. This is because a feature representation $\phi_s(p)$ can generate better gradients $\frac{\partial \phi_s}{\partial \hat{p}}$. The reconstruction loss $L_{s \to t}$ is adjusted accordingly to support the use of features ϕ:

$$L_{s \to t} = \sum_{p} pe\left(\phi_s(\hat{p}), \phi_t(p)\right) \tag{3.16}$$

3.4.2.2 Image Reconstruction Loss

The single-image reconstruction loss is a standard loss function used in auto-encoder networks. It is specified as the $L1$ distance between the input and the reconstructed image after being sent through the auto-encoder:

$$L_{rec} = \sum_{p} |I(p) - I_{rec}(p)|_1 \tag{3.17}$$

3.4.2.3 Discriminative Loss

The discriminative loss encourages the learned features ϕ to have large gradients. Gradients from the image itself are used to emphasize the low-texture regions that receive large weights:

$$L_{dis} = -\sum_{p} e^{-|\nabla^1 I(p)|_1} |\nabla^1 \phi(p)| \tag{3.18}$$

Where ∇^1 is the first-order derivative $\nabla^1 = \partial_x + \partial_y$ with respect to the image.

3.4.2.4 Convergent Loss

The convergent loss L_{cvt} is used to encourage smoothness in the feature gradients. The smoothness ensures that the gradients are consistent during the optimization steps by penalizing the second-order gradients with the $L1$ distance between the gradients and the features. This loss can be viewed in the same way as Manydepth and Monodepth2's smooth loss, which is further explained in Section 3.3.2 and Equation (3.9):

$$L_{cvt} = \sum_{p} |\nabla^2 \phi(p)|_1 \tag{3.19}$$

Where ∇^2 is the second-order derivative $\nabla^2 = \partial_{xx} + 2\partial_{xy} + \partial_{yy}$ with respect to the image.

3.4.2.5 Final Loss for the Auto-encoder Network

The final loss value for the auto-encoder is a weighted combination of the pre-mentioned regularizers:

$$L_s = L_{rec} + \alpha L_{dis} + \beta L_{cvt} \tag{3.20}$$

Where the weights α and β are both set to $1.0 \cdot 10^{-3}$, found through cross-validation.

3.4.3 Feature-metric Loss

The feature metric loss is one of the novel ideas presented by Feature Depth. The idea is to use the L1 distance between the features generated after running I_s and I_t through the FeatureNet encoder:

$$L_{fm} = |\phi_s(\hat{p}) - \phi_t(p)|_1 \tag{3.21}$$

3.4.4 Final Loss

As established earlier, the final value for the multi-view reconstruction problem is the photometric loss pe defined in equation Equation (3.7) and the novel feature metric loss:

$$L_{s\to t} = L_{ph} + L_{fm} \tag{3.22}$$

The use of minimum photometric error rather than using the average, as per [29, 47] is also adapted here, leaving the final multi-view reconstruction loss as:

$$L_{s\to t} = \sum_{p} \min_{n} L_{t+n\to t}\left(\phi_{t+n\to t}(\hat{p}), \phi_t(p)\right) \tag{3.23}$$

The final loss for the entire architecture is a combination of the multi-view and single-view reconstruction loss:

$$L = L_s + L_{s\to t} \tag{3.24}$$

3.5 Vision Transformers for Dense Predictions

This section will take a deep dive into the Dense Prediction Transformer (DPT) by Ranftl *et al.* [42] and look at how the Vision Transformer from [17] was modified to perform dense prediction tasks such as monocular dense depth estimation and semantic segmentation. This section will focus on the monocular dense depth estimation use case, as this thesis primarily focuses on this task. However, the semantic segmentation task is discussed in Chapter 5 and Chapter 6.

3.5.1 Architecture

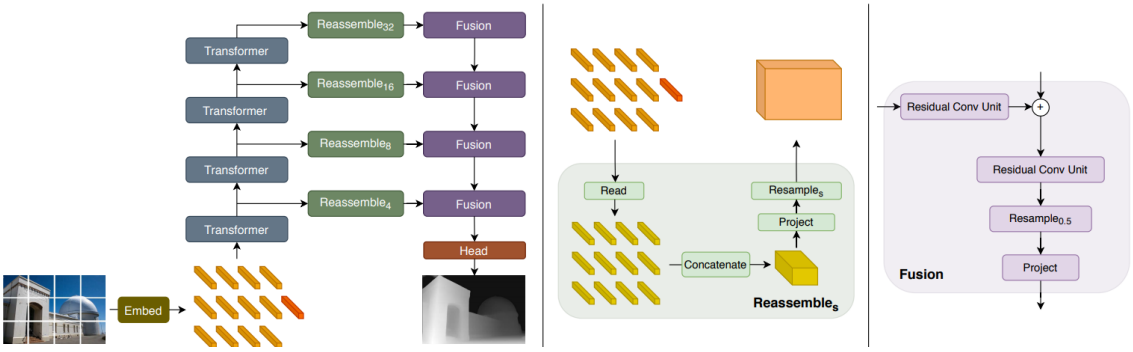


Figure 3.5: Illustration of DPT's final architecture. Adapted from Ranftl *et al.* [42]

DPT's architecture can be seen in Figure 3.5. The process starts with an input image divided into patches of size $p = 16$, much like with the Vision Transformer (ViT). These patches are embedded into features and flattened and encoded with a positional value representing the patch's original position. A standalone readout token is also added, and these tokens are sent to a set of transformers. At four different points between two transformer outputs, the output

features are sampled and reassembled at four different resolutions into feature maps depending on the architecture used. These feature maps are progressively fused before, finally, being fed into a task-specific head network. This head network can be trained to perform any dense prediction task, e.g., monocular depth estimation and semantic segmentation.

3.5.2 Transformer Encoder

The main component of DPT's encoder is a vision transformer, discussed in Section 2.2.3. The input images fed to the DPT are preprocessed similarly to a vision transformer and consist of using image patches embedded into a feature space, or as an alternative, features generated from feeding the images through the layers of a ResNet50 network. These patches are flattened, embedded with a positional encoding, and fed into the transformer. As per ViT, a unique standalone token, called the class token in ViT, is added at this point. For this use case, this token is called the *readout* token.

Ranftl *et al.* [42] propose three different transformer-based encoder architectures for DPT: A base model, called ViT-base, which uses the patch-based embedding procedure and 12 transformer layers, An extended version of the base model, called ViT-large, using the same embedding, but 24 transformer layers, and a hybrid model, called ViT-hybrid, which uses the resulting features after sending the image through a ResNet50 model [10]. The output from the ResNet model is used as the embeddings for a set of 12 transformer layers. The patching dimension uses $p = 16$ for all models, following [17].

After applying the embedding procedure to an image of dimension $H \times W$ pixels, one is left with $t^0 = \{t_0^0, ..., t_{N_p}^0\}, t_n^0 \in \mathbb{R}^D$ tokens, where $N_p = \frac{HW}{p^2}$. t_0 refers to the readout token, and D is the dimension of each feature token, where $D = 768$ for ViT-base and ViT-hybrid, and $D = 1024$ for ViT-large. For ViT-base and ViT-large, as there is no compression of the pixels, and $D > p^2, p = 16$, this results in no compression. Thus, pixel-level accuracy input is kept. For the ViT-hybrid, the features are at a $\frac{1}{16}$ resolution, which still is greater than other more standard convolutional-based encoders.

The tokens are sent through the transformers, where one transformer consists of feed-forward nets and multi-headed attention blocks. The output from a transformer at layer l is referred to as t^l, where the input to layer l is the output $t^{l-1}, l-1 \geq 0$. In a standard vision transformer, all output except for the CLS token is dropped in the final decoding layer. However, this is not the case for the DPT, as there is a one-to-one correspondence between the output and the inputted image patches. Hence, all tokens are kept in the output.

3.5.3 Convolutional Decoder

After being sent through the transformer layers, one is left with a set of tokens from the transformer layers t. These tokens need to be decoded into image-like feature representations and finally fused into a dense prediction. Ranftl *et al.* [42] propose a three-stage *Reassemble* operation for this task, which generates a feature map from the tokens. This map is fed into a fusion block that progressively upsamples the representation to half the size of the input image. Finally, the fused feature map is sent to a task-specific head network.

3.5.3.1 Reassaemble Operation

The reassemble operation can be defined as follows:

$$\text{Reassemble}_s^{\hat{D}}(t) = \big(\text{Resample}_s \circ \text{Concatenate} \circ \text{Read}\big)(t), \tag{3.25}$$

Here, s denotes the output size ratio of the recovered representation with respect to the input image. $\hat{D}$ represents the output feature dimension. t is the output from a transformer layer.

Read Operation The Read operation is responsible for handling the information in the readout token and effectively reduces the number of dimensions by 1. Due to the readout token not being actively used in this dense prediction task, this token can either be ignored by dropping it, added to all other tokens, so that the new tokens are $\{t_1 + t_0, ..., t_{N_p} + t_0\}$ or concatenated with the other tokens before projecting the representation to the original dimension size D with a multi-layered perceptron. Doing this leaves the read operation as:

$$\text{Read}_{proj}(t) = \{\text{mlp}(\text{cat}(t_1, t_0)), ..., \text{mlp}(\text{cat}(t_{N_p}, t_0))\} \tag{3.26}$$

Concatenate Operation The concatenate operation reshapes the resulting tokens into an image-like representation. This reshaping is done using the tokens' positional encoding and placing them according to their original position. It is defined as:

$$\text{Concatenate} : \mathbb{R}^{N_p \times D} \rightarrow \mathbb{R}^{\frac{H}{P} \times \frac{W}{P} \times D} \tag{3.27}$$

Where P refers to the number of tokens, and thus also the number of patches, as there is a one-to-one relationship between input patches and output tokens.

Resample Operation The last operation in the reassemble operation is a resampling operation that scales the representation:

$$\text{Resample}_s : \mathbb{R}^{\frac{H}{P} \times \frac{W}{P} \times D} \rightarrow \mathbb{R}^{\frac{H}{s} \times \frac{W}{s} \times \hat{D}} \tag{3.28}$$

This operation is implemented by using a convolutional block. The block consists of a 1×1 convolution to project the input to $\hat{D}$, followed by a strided 3×3 convolution or transpose convolution, depending on if $s \geq p$ or $s \leq p$ to implement spatial down and upsampling which is used respectively.

3.5.3.2 Fusion Operation

A RefineNet-based feature fusion block by Lin *et al.* [60] inspires the fusion operation. This block progressively upsamples the output from the reassemble operation by a factor of two in each fusion block, leaving the final fusion block output with half the height and width of the original input image.

The reassemble and fusion operations are used at the output from four different transformer layers and with four different resolutions scaled by s. ViT-large uses layers $l = \{5, 12, 18, 24\}$, ViT-base uses $l = \{3, 6, 9, 12\}$. ViT-hybrid uses the first and second ResNet block together with

layers $l = \{9, 12\}$. The default architecture uses the Read_{proj} read operation in the reassembling and produces feature maps with a dimension $\hat{D} = 256$. These final architectures are named DPT-Base, DPT-Large, and DPT-Hybrid, respectively.

3.5.4 Monocular Dense Depth Estimation Head

The head attached to the output from the last fusion block is a simple sequential network consisting of a single bilinear upsampling layer and a couple of convolutional layers. There no need for anything more here due to the final fusion layer already being half the dimension of the original image. There is little to no information regarding the loss used when training this network, as the complete code set for the DPT is not publicly available yet, but one can assume that it follows [61]'s scale and shift-invariant losses due to [61] being created by virtually the same people from Intel Labs that created DPT.

3.5.5 Training

Transformers need tremendous amounts of data to be able to show their proper potential performance. Ranftl *et al.* addresses this by using their MIX5 dataset and five other large datasets containing disparity and segmentation data to create a new dataset called MIX6. This dataset contains a total of 1.4 million samples with associated depth and disparity maps. As a pretraining measure, they train with a specially selected subset of the dataset for 60 epochs before starting training with the complete dataset. With this, they train the decoder and encoder with an Adam optimizer. The encoder is trained with a learning rate of 1e-4 and has pretrained weights from ImageNet. The decoder is trained with a learning rate of 1e-5 and has randomly initiated weights.

3.6 Data Extraction and Preparation

The depth detection systems are all set up to use the KITTI dataset, as this is the most used benchmark dataset for depth detection. However, to verify that the depth detection systems can generalize to other data, it is interesting to see how they perform on other datasets. This thesis will focus on the Lyft and DDAD datasets, which are introduced in Section 2.7. This section will discuss how the dataset was collected and prepared to be used in the dense depth detection systems.

3.6.1 Data Organization

Due to more than one dataset being used in this thesis when training Manydepth and Feature Depth, it would be beneficial to establish a shared way of organizing the data for all the datasets. All the datasets are already defined as scenes consisting of data points covering a recording period, either in a separate lookup table of IDs or physically in separate folders. As both Manydepth and Feature Depth need to relate to the current frame's past and future frame during a training step, keeping this structure for the other datasets is beneficial.

3.6.1.1 Observations

Lyft, DDAD, and the NAP-lab dataset all have a unique intrinsic matrix associated with a scene, unlike the KITTI and Cityscapes dataset, which only uses a single static intrinsic matrix. Therefore, this intrinsic matrix needs to be available during training. All the datasets also have data available from multiple images and lidars. Thus, data points should be hierarchically separated to support using data from multiple sensors simultaneously.

3.6.1.2 Suggested Requirements

The following decisions were made regarding the data organization setup, considering the observations in the previous section:

The files will be separated into folders; hereafter called a scene, rather than a lookup table In the PackNet codebase, TRI uses its own Data Governance Policy (DGP) tools in the dataloader itself. While this is convenient for an already established way of organizing data, it is not very useful for datasets with other governance policies. A valid option is to develop such a tool for this task, but as the tool would need to take a stance on its internal data organization, this would result in a repeated workload, which could be a bit excessive only for this thesis.

Each scene's folder will contain subfolders, where each subfolder contains data for one and only one sensor The possibility of using a lookup table for organizing the files was considered. However, this was considered to defeat the purpose of using separate scene folders in the beginning. Additionally, it would result in a more unorganized setup in the scene folders as well. It is important to note that one scene can contain data points for more than one sensor, e.g., a front-facing and backward-facing camera.

A separate, precomputed file containing the intrinsic matrix for each camera with extracted data in the scene exists in the scene's folder The file containing the intrinsic matrix needs to have a name identifiable with the subfolder's name containing the images of the associated camera. It was considered to include the calibration files instead. However, all the datasets have a different form of calibration files, and handling these differences in a dataloader unnecessarily complicates the dataloader. Another weakness of attaching the calibration files is that this would result in the dataloaders being dependent on calibration files. This behavior is unwanted and unnecessary as it is already known what kind of data is needed to train the methods, and there is no point in adding data that is not important in a scene folder.

3.6.1.3 Final Data Organization Setup

The final data organization can be viewed in Figure 3.6. Formally speaking, A dataset is divided into scenes S. Each scene $s \in S$ consist of data points P, where a data point $p \in P$ is defined as $p = \{d_0, ...d_x\}$. Here d_i represents the data readings for a sensor $i, 0 \leq i \leq x$, where x is the total number of sensors. Data from a sensor d_i exist in its own subfolder, and contains $d_i = \{r_0, ..., r_y\}, y = (n \cdot \text{sampling rate})$ sensor readings. The relative index y points to the same

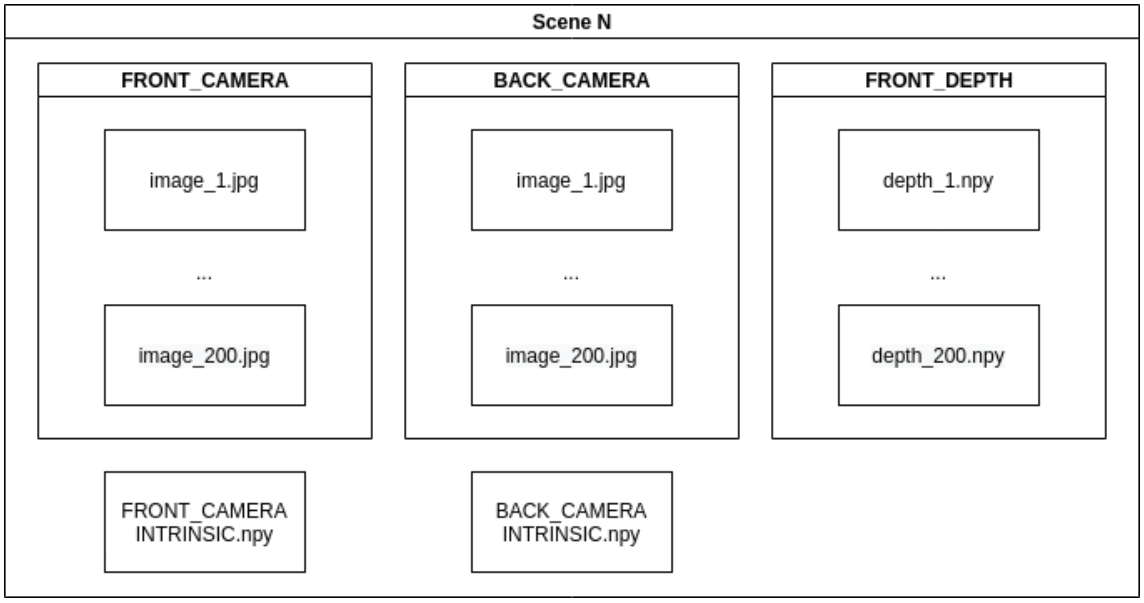


Figure 3.6: The final dataset organization setup. This illustrates a single scene s in the dataset. A complete dataset consists of multiple scenes like this

point in global time for all the sensors D. For a scene s, there exist j precomputed intrinsic files, where $j =$ number of cameras in the scene.

3.6.2 Lyft

The Lyft dataset is available under a public license and is shipped as tarfiles, one for training data and another for test and validation data. Both files are approximately 58GB and contain camera data from 5 different cameras and two lidars, with 22 000 sensor readings per sensor. These sensor readings are from Lyft's fleet of autonomous vehicles, captured in sequences of 25-45 seconds. The images have a dimension of 1920x1080 for some sequences and 1224x1024 for others, with the majority of the images being in the latter format, and they are compressed with JPEG compression. All images are placed in a shared folder. The images are named with unique IDs generated by a generator. All the images are synced up with lidars found on the vehicle's top and in the vehicle's front. Data from these lidars are saved as binary files as points in the format of $\{x, y, z, intensity, ring\ index\}$. The sensor layout can be viewed in Figure 3.7

3.6.2.1 Data Preparation

Due to the images being placed in a single folder, they had no reference to which sequence they belonged. The depth detection methods all rely on knowing the images' temporal context, and without knowing this, they could not be used by depth detection methods. Therefore, they needed to be moved into separate folders, one for each sequence. Another necessary preparation was to extract the correct intrinsic parameters for a sequence, as the images are collected from multiple vehicles.

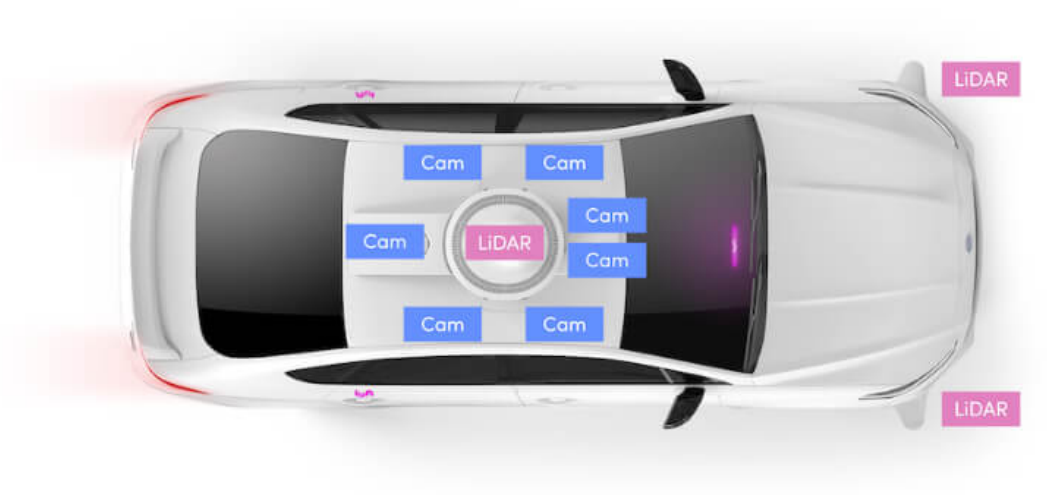


Figure 3.7: Sensor layout on Lyft's vehicles. The image shows the sensor layout on one of the vehicles in Lyft's fleet. The sensors utilized here are the front and backward-facing cameras and the top-mounted lidar. Adapted from Kesten *et al.* [49]

3.6.2.2 Generating Extra Data

One of the best ways of improving deep learning-based systems is to provide them with more data. As the pose network used in these depth detection methods expects the views shown during training to have consistent motion, one can not utilize data from cameras pointing in other directions. However, one can artificially create more data by using the backward-facing camera in reverse temporal order. By doing so, the pose will change according to the front-view camera, and one has, in theory, duplicated the amount of available data. However, it is worth noting that this is not tested and verified. Utilizing this data will be conducted as a separate experiment, and the results are discussed in the results and discussion sections.

3.6.2.3 Depth Maps

The proposed depth detection methods are based on self-supervised learning and will only use the camera data as input. However, to surveil how the methods are performing during training, one can utilize the data from the lidars when performing validation. The depth detection methods are already set up to use a depth image, where the lidar points are projected onto the camera plane. First off, the ring index and intensity can be dropped, as this is not interesting for this use. This results in a list $\boldsymbol{P}$ consisting of N points $\boldsymbol{p_n} = [x_n, y_n, z_n]$. These points exist in the lidar's frame of view and were sampled at time j. To use these points in a depth map, they need to be transformed to the camera's frame and to the time of sampling the image, at time i.

The transform from the lidar frame at time j to the camera frame at time i is described as follows:

$$T_{L_j}^{C_i} = T_{V_i}^{C_i} T_W^{V_i} T_{V_j}^W T_{L_j}^{V_j} \tag{3.29}$$

Where T_A^B is the transformation matrix describing the transform from frame A to frame B, the subscript i and j represent the time of capturing the camera data and the lidar data, respectively.

The transformation matrices are:

- $T_{L_j}^{V_j}$ is the transform from lidar frame at time j to the vehicle frame at time j
- $T_{V_j}^W$ is the transform from the vehicle frame at time j to the world frame.
- $T_W^{V_i}$ is the transform from the world frame to the vehicle frame at time i
- $T_{V_i}^{C_i}$ is the transform from vehicle frame at time i to the camera frame at time i

This transformation can be used on the points in the lidar frame and transform them to the camera frame:

$$\boldsymbol{P}^{C_i} = T_{L_j}^{C_i} \boldsymbol{P}^{L_j} \tag{3.30}$$

The points can be projected to the camera plane by taking the dot product of the points and the camera intrinsic matrix:

$$\boldsymbol{P} = K\boldsymbol{P}^{C_i} \tag{3.31}$$

Here, K is the 3x3 camera intrinsic matrix with an added dimension to support a homogeneous transform, making K a 4x4 matrix.

The points $\boldsymbol{P}$ are now mapped to the camera plane. The x and y coordinates represent the x and y coordinate on the camera plane. The original Z values from $\boldsymbol{P}^{C_i}$ is assigned to be the original depths:

$$\boldsymbol{P}_z := \boldsymbol{P}_z^{C_i} \tag{3.32}$$

A depth image can be generated with the mapped points and depths by creating an empty image of the same dimension as the original image and use the z value on all available points x and y from $\boldsymbol{P}$. It is worth mentioning that $\boldsymbol{P}$ does not cover points for all x and y values in the original image. Points that have no accompanying z are left as 0. These depth maps can be loaded at the validation step and used to calculate RMSE, Log RMSE, Absolute and Square Relative Error. These metrics are described in Section 3.8.2

3.6.2.4 Cropping

The dimension of the images used is 1224x1024. This close-to-squared dimension results in the network trying to predict depth for many unnecessary areas. Examples are depth detection to static objects in the scene like the car itself and the sky. Therefore, the images' height was cropped with 300 pixels on the top and bottom, leaving the final image dimension to be 1224x424. When the images are cropped like this, one must also modify the principle point in the intrinsic matrix, defined in Equation (3.1), accordingly. The new principle points are

$$\hat{c}_x = c_x - \text{pixels cropped from the left} \tag{3.33}$$

$$\hat{c}_y = c_y - \text{pixels cropped from the top} \tag{3.34}$$

Resulting in a new intrinsic matrix K:

$$K = \begin{bmatrix} f_y & 0 & \hat{c}_x \\ 0 & f_y & \hat{c}_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.35}$$

3.6.2.5 Extraction

Lyft has created a python package containing all metadata to the dataset. This metadata included which images belonged to which sequences, their respective intrinsic parameters, and the correct order of the images in a sequence. With this information, the images were extracted to separate folders, one per sequence. The folder contained all relevant images, where the images were renamed to contain their correct index in that specific sequence and the correct intrinsic parameters for that sequence. A separate list containing lists of information on the relative path to the sequences and the image indices was generated during this process, as the Monodepth2 system requires this format. This list was split into two separate lists for training and validation data, with a 5% split ratio. The list was flattened to contain only paths and indexes, leaving the final data to be 17 000 training and 700 validation images for each camera. This results in the potential of having 34 000 training images and 1600 validation images, all with accompanying depth images if the backward-facing camera data is utilized. After cropping and resizing, the final image size was 608×224.

3.6.3 DDAD

3.6.3.1 Data Collection

The DDAD dataset is available under a Creative Commons License and is shipped as one 257 GB tar file that contains both training and validation data. The dataset contains readings from seven cameras and four lidars. Their placements are seen in Figure 3.8. The data is collected as scenes consisting of 5 to 10 seconds of data, resulting in 50 to 100 frames. The dataset contains 150 training scenes and 50 validation scenes, translating to 12 650 training frames and 3 950 validation frames. However, due to being limited on the amount of available data here, the total number of validation frames was reduced to 900.

All sensor readings for a given scene exist in their folder, but the file names have no information regarding the temporal order of the data. All the sensor readings are synced and run

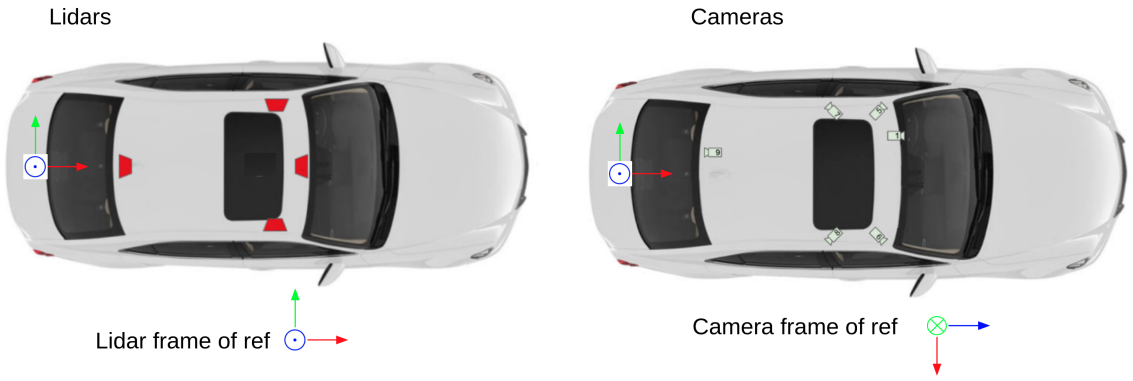


Figure 3.8: Sensor layout on DDAD's vehicles. The image shows the sensor layout on one of the vehicles in fleet used when collecting the DDAD dataset. The sensors utilized here are the front-facing cameras and the top-mounted lidar. Adapted from Guizilini *et al.* [30]

on a 10Hz interval. The cameras produce images with a dimension of 1936x1216 and are already rectified. The lidar data are saved as serialized NumPy arrays. All sensors are calibrated and have associated extrinsic parameters found in the scene folder. The cameras also have an associated intrinsic matrix.

3.6.3.2 Data Preparation

The Toyota Research Institute has created its own Dataset Governance Policy (DGP), which contains systems that ensure traceability, reproducibility, and standardization for all their datasets. This policy was used when extracting the DDAD data. The training and validation datasets have an associated JSON file containing information about the file names' temporal order. When this JSON file is fed to the DGP systems, it is possible to get the temporal order of the files. The DGP system can also generate depth images for all cameras with data from the lidars, already on the format discussed in Section 3.6.2.3. The DDAD dataset also has a backward-facing camera, but using the backward-facing camera as extra data will be limited to the Lyft dataset.

Due to the images having a lot of uninteresting space, the images were cropped accordingly to Section 3.6.2.4, and the final dimension is 1936x616. Following Lyft, the images was also resized, so that the final image size is 608×224.

3.6.4 KITTI

3.6.4.1 Data Collection

The KITTI data were collected by following the instructions found in the Manydepth repo's reference to Monodepth2's instructions. These instructions specify which KITTI archives to download and how they should be unpacked and compressed.

3.6.4.2 Data Preparation

In general, there was little to no preparation needed with the KITTI data, as its default layout is the layout used as a base when creating the depth detection methods. The only preparation

done is to compress the data with JPEG compression. The depth detection repositories also contained the training and validation split used, a subset of the KITTI dataset explicitly selected to be used as monocular depth prediction training by Zhou *et al.* [44]. The final images used had a dimension of 640 × 194.

3.6.5 NAP-lab

3.6.5.1 Data Collection

The videos of the NAP-lab vehicle in action were collected with NAP-lab's vehicle. It is equipped with eight cameras giving a 360-degree field of view around the car, one lidar mounted on the top of the vehicle and one at the vehicle's front, and a lidar mounted on the rear right side of the vehicle. The 60-degree field-of-view camera is utilized in this thesis as the only input sensor.

Figure 3.9: The NAP-lab Vehicle. NAP-lab's vehicle is a Kia e-Niro equipped with sensors, a drive-by-wire-kit, and computational hardware. This equipment makes it possible to use this vehicle as a platform to develop software for autonomous vehicles

3.6.5.2 Missing Calibration

The SSL-based methods utilize projection both during training and testing. Therefore, a complete set of intrinsic parameters, as discussed in section 2, is needed to perform both training and predictions. Unfortunately, when looking in the provided calibration files, the parameters f_x and f_y containing information about the camera's focal length measured in pixels were missing. These parameters were missing due to NAP-lab utilizing the NVIDIA Driveworks stack, which uses a distortion model, *f-theta* [62], rather than using the, more commonly used, pinhole projection model. However, using the knowledge from Section 3.2, as the camera type

and lens type were known, the f_x and f_y parameters could be calculated using the focal length and pixel size, measured in real distance.

3.6.5.3 Data Preparation

Due to the NAP-lab data becoming available at a late stage of the thesis, it was decided that the NAP-lab data should be structured in the same way as the Lyft and DDAD dataset to save time. The collected data existed as mp4 and h264 video files with a framerate of 30 FPS. In addition, all the videos had accompanying JSON files containing the calibration data. All the videos were given a folder, where a folder contained a subfolder with the video's associated frames, where a frame had the dimension of 1920×1208. In addition, a separate file was added to the folder containing the associated camera's intrinsic matrix. The c_x and c_y parameters were collected directly from the JSON file to construct the intrinsic matrix. The f_x and f_y parameters needed to be calculated. As mentioned, information about the camera and lens type was available. This data indicated that the focal length was 5.49mm and the pixel size were $3\mu\text{m} \times 3\mu\text{m}$. f_x and f_y was calculated to be:

$$f_x = f_y = \frac{5.49 \cdot 10^{-3}}{3.0 \cdot 10^{-6}} = 1830.0 \tag{3.36}$$

Like for the Lyft dataset, these images were also cropped to remove some of the unnecessary information. The images were cropped with 250 pixels from the top and bottom, leaving the final image dimension to be 1920×708. After resizing, the final image size is 608×224, following Lyft and DDAD. The intrinsic matrix is cropped in the same way as in Section 3.6.2.4, leaving the final intrinsic matrix to be:

$$K = \begin{bmatrix} 1830.0 & 0 & c_x \\ 0 & 1830.0 & c_y - 250 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.37}$$

Here, c_x and c_y is extracted from the associated video's calibration file.

3.7 Modifictions of the Methods

After choosing to use and train the ManyDepth [47] and Feature Depth [46] systems in Section 3.1, these methods needed some modifications and additions to be able to utilize the datasets also chosen in the same section. This section documents these modifications and additions are made to be able to use them with the Lyft, DDAD, and NAP-lab datasets.

3.7.1 Dataloaders

Both Feature Depth and Manydepth were already set up to use data from KITTI and Cityscapes. As these dataloaders do not out-of-the-box support the format for the Lyft, DDAD, and NAP-lab datasets, custom dataloaders were needed. These three datasets were all extracted to the layout described in Section 3.6.1, and could therefore utilize close-to-the same dataloader setup.

After having created the dataloaders for Manydepth, it was discovered that Feature Depth used virtually the same dataloader superclass, as both dataloaders are based on the format of Monodepth2 [28]. Therefore, a dataloader for a dataset could be used for both methods with minor changes.

One important feature that is not available in the KITTI and Cityscapes dataloaders is to crop the input data correctly. As mentioned in Section 3.6.2.4, the intrinsic matrix would also need to be modified when cropping the images. Another feature that was is not implemented but is supported is the loading of different intrinsic matrixes depending on which video sequence the frames belong to, as all the datasets being used here have a unique intrinsic matrix per scene. In the custom dataloaders, one can specify the number of pixels to crop from all sides, and the intrinsic matrix will be calculated according to Equation (3.1).

Another problem with the existing dataloaders was that they used the point cloud data and extrinsic calibration data in the dataloaders and transformed it into a depth map during training. These depth maps had already been precomputed, discussed in Section 3.6.2.3. Therefore, the transformation step and the need for the extrinsic data could be dropped, as the depth images could be loaded directly.

3.7.2 Supporting Custom Depth Maps

While testing the newly created dataloaders, it was discovered that both Manydepth and Feature Depth crops the depth maps in a separate part of the codebase. This cropping is done to match the cropping setting done by Eigen *et al.* [37] and Garg *et al.* [63] and applies only to the KITTI and Cityscapes data but not to custom datasets. Consequently, this led to the cropping being moved into the KITTI and Cityscapes dataloaders so that the codebase could support custom datasets that have accompanied datasets.

3.7.3 DPT

The DPT has code available for performing predictions. However, the code for training is not yet published. Some pretrained models exist, and for this use case, the pretrained model that is fine-tuned on the KITTI dataset will be used.

3.8 Training Details

This section contains some additional information regarding what type of computing hardware was used to train the methods and which evaluation metric was used for the online evaluation.

3.8.1 Computing Hardware

The training was mainly done on the Oppdal cluster with NVIDIA Tesla T4 GPUs, which have 16 GB of available video memory. Manydepth only utilizes a single GPU at the time, but Feature Depth can utilize multiple. Training with Feature Depth was done with up to 10 T4 GPUs. The IDUN computing cluster [64] was also used, with V100 and P100 GPUs.

3.8.2 Evaluation and Metrics

3.8.2.1 RMSE

Root Mean Square Error is a commonly used metric for measuring the difference between two values. RMSE compares predicted depth values to ground truth values generated from separate depth sensors, often lidar maps. It is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{|T|}\sum_{y\in T} \|\log y - \log y^*\|^2} \tag{3.38}$$

3.8.2.2 Absolute and Square Relative Error

Absolute and Square Relative Error is error measurements used to a ratio that describes the normalized values of the actual error. It is used to describe a relative number rather than the actual difference, as this can vary from predictor to predictor. A perfect model would result in the ratio equalling 0. The errors are defined as follows:

$$\text{Abs rel err} = \frac{1}{|T|}\sum_{y\in T} \frac{|y - y^*|}{y^*} \tag{3.39}$$

$$\text{Squared rel err} = \frac{1}{|T|}\sum_{y\in T} \frac{\|y - y^*\|^2}{y^*} \tag{3.40}$$

3.8.2.3 Scale-Invariant Error

The scale-invariant error is an error metrics proposed by Eigen *et al.* [37]. As standard metrics tend to be biased towards how close the mean depth predicted is to the ground truth, they are not necessarily the best way of comparing depth models. Eigen *et al.* proposed an error metric that measures relationships between points in a depth map that is irrespective of the absolute global scale of the map. The metrics have become a standard error function for depth estimation systems. It is defined as:

$$D(y, y^*) = \frac{1}{n}\sum_{i=n}^{n} (\log y_i - \log y_i^* + \alpha(y, y^*))^2 \tag{3.41}$$

where α is:

$$\alpha = \frac{1}{n}\sum_{i=n}^{n} (\log y_i^* - \log y_i) \tag{3.42}$$

Chapter 4

Experiments and Results

Four main experiments were conducted as a part of the work to answer the research questions. The experiments mainly consist of training the selected methods on the chosen datasets and observe the results. For all the experiments, it is essential to take note of any mysterious and unexpected behavior and try to understand why this behavior occurs.

The following experiments are proposed:

0. Benchmark Manydepth and Feature Depth on the KITTI dataset as a benchmark to compare the other datasets with.

1. Train Manydepth and Feature Depth on the Lyft dataset. Compare the validation segments with DPT.

2. Train Manydepth and Feature Depth on the DDAD dataset. Compare the validation segments with DPT.

3. Test if extending a dataset with a camera from the backward-facing camera improves the performance of the best performing method from experiments 1 and 2.

4. Train the best-performing method from experiments 1 and 2 on data from the NAP-lab vehicle. Compare the validation segments with DPT.

The first proposed experiment is not as much an experiment in itself. The reason for having this experiment is to verify the results of the proposed methods and confirm that training them with the KITTI dataset matches their reported results. Therefore, this experiment is considered to be a kind of "pre-experiment."

4.1 Experiment Setup and Description

By default, the experiments will use data from a single, forward-facing camera to ensure equal terms between the dataset unless anything other is explicitly stated. Unlike Zhou *et al.* [44] and Eigen *et al.* [37], which remove frames with no motion, all the available frames from the front-facing cameras will be used.

Each experiment will present the results from the best epoch of training, with the primary evaluation metric being the absolute relative difference, following Monodepth2 [29], Manydepth [47], PackNet [30], Feature Depth [46], and DPT [42], and other SotA SSL-based methods. Additional results like plots of the evaluation metrics and different loss values will also be presented in a separate result section. The experiments will have a section dedicated to discussing the results, with a more in-depth and further analysis in Chapter 5.

In the Lyft and DDAD datasets, there exists a depth map associated with each training image. These depth maps are only used to perform an online evaluation, as they are needed to calculate the evaluation metrics described in Section 3.8.2 of the methods and never as input due to the methods being self-supervised.

4.2 Codebases

The codebases created by the authors of the selected methods were used to ensure comparable results to the results presented in the papers. This section describes the details about the datasets of the selected methods.

4.2.1 Manydepth

All the experiments training Manydepth utilizes the codebase created by Watson *et al.* and can be found here The modifications done and the description of the data used can be found in Section 3.7. An in-depth description of Manydepth can be found in Section 3.3.

4.2.1.1 Training Parameters

Manydepth was trained on a single NVIDIA T4 GPU with a batch size of 12. An Adam optimizer was utilized, with a learning rate of $1.0 \cdot 10^{-4}$ for the first 15 epochs and dropping to $1.0 \cdot 10^{-5}$ for the last five epochs. The frames indexed at $+1$, and -1 relative to the current frame were used for future and past frames. After epoch 15, the weights for the pose and consistency network were frozen together with the adaptively learned d_{min} and d_{min}, so that the depth model could be fine-tuned with a scene with no motion. With this, the model only focuses on reprojection training in the last five epochs. Weights pretrained on ImageNet are used in all the ResNet-based networks.

4.2.2 Feature Depth

The codebase created by Shu *et al.* was used when training Feature Depth with the modifications described in Section 3.7 added. The inner details about Feature Depth can be found in Section 3.4. The codebase can be found here.

4.2.2.1 Training Parameters

Due to the feature depth model using ResNet50 rather than ResNet18 as Manydepth, the model requires more memory per training iteration than Manydepth. All three networks were trained

simultaneously on 10 NVIDIA T4 GPUs, where each GPU used a batch size of 4. An Adam optimizer was utilized with the initial learning rate set to $1.0 \cdot 10^{-4}$ before being halved in epoch 20 and 30. The discriminative weight α, convergent weight β weights described in Section 3.4.1.1 were all set to $1.0 \cdot 10^{-3}$, following the recommendations from the Feature Depth paper [46]. As for the KITTI dataset, the d_{min} and d_{max} were set to 0.1 and 80.0, respectively. Weights pretrained on ImageNet are used in all the ResNet-based networks.

4.2.3 DPT

Initially, training DPT would be conducted as a separate experiment, as Ranftl *et al.* reported that they would release the training code in March. Currently, their models and prediction code have been released, but the training code has yet to be released. Hence, DPT will not be trained. However, it was decided that an experiment using DPT should be conducted nevertheless. This experiment will use the DPT-Hybrid model with weights pretrained on the KITTI dataset. This model will be used to run predictions on segments from Lyft, DDAD, and NAP-lab. DPT's codebase containing code for performing predictions can be found here.

4.3 Experiment 0: Benchmarking with KITTI

4.3.1 Setup

The KITTI dataset was extracted in the format reported for Manydepth [47], and Feature Depth [46], and will not be following the setup in Section 3.6.1, due to the codebases already containing dataloaders for the KITTI dataset. Both Manydepth and Feature Depth utilize a subset of the KITTI dataset specifically sampled to remove scenes with no motion and other vehicles moving with the same relative velocity as the ego-vehicle. This subset was first introduced by Eigen [37], and further improved by Zhou [44]. It contains roughly 40 000 frames, where each frame has an associated depth image generated by a lidar.

4.3.2 Results

4.3.2.1 Manydepth

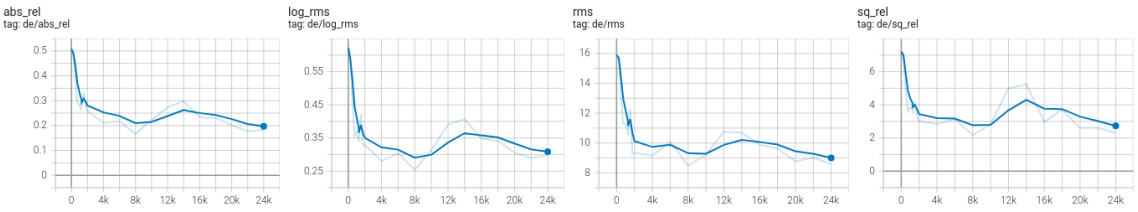
Manydepth was trained for 40 000 training iterations divided into 20 epochs. The model achieved an absolute relative difference score of 0.104. The loss progression can be seen in Figure 4.1, the qualitative results in Figure 4.3, and the evaluation metrics in Table 4.1

4.3.2.2 Feature Depth

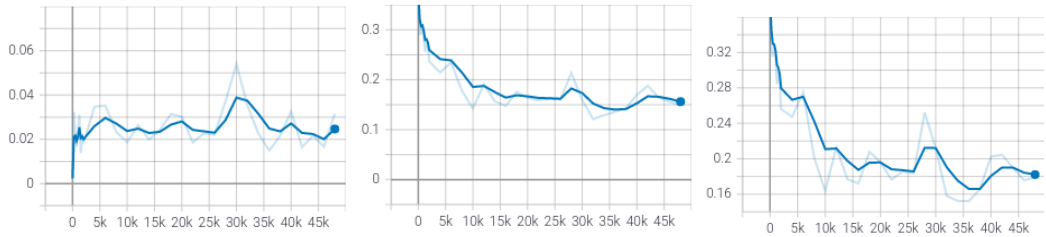
Feature Depth were improving up until the last epoch. achieving an absolute relative difference of 0.099. The loss plots can be found in Figure 4.2, qualitative results in Figure 4.3 and the best performing model's evaluation metrics in Table 4.1

4.3.3 Discussion

Both methods perform as expected compared to the reported values in their respective papers. This result indicates that the KITTI dataset is fit to use as an ideal representation of how loss

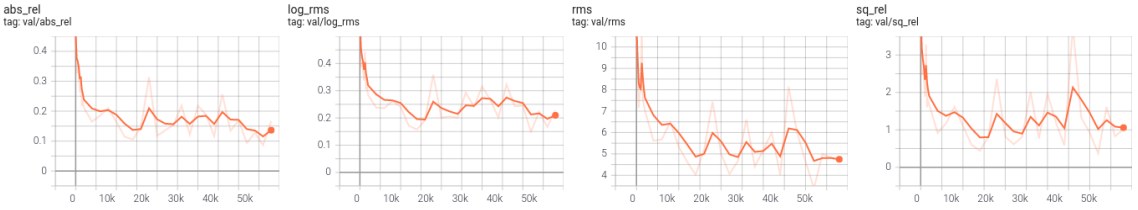


(a) Evaluation Metrics

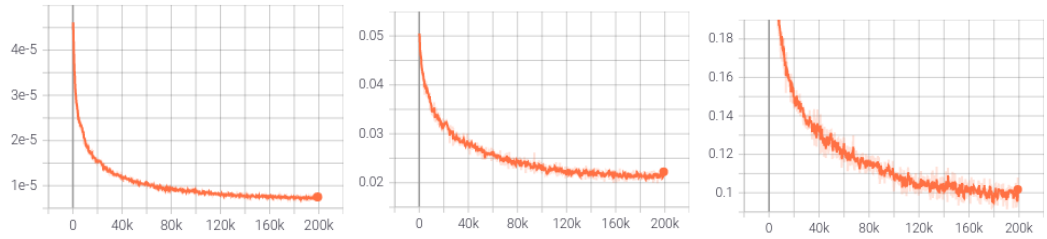


(b) Consistency Loss **(c)** Reprojection Loss **(d)** Total Loss

Figure 4.1: Experiment 0: Manydepth KITTI results. Results after training Manydepth for 20 epochs with the KITTI dataset



(a) Evaluation metrics



(b) Feature-metric Loss **(c)** Reconstruction Loss **(d)** Total Loss

Figure 4.2: Experiment 0: Feature Depth KITTI plots. Results after training Feature Depth for 40 epochs on the KITTI dataset

Dataset Name	FPS	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Manydepth	27	0.104	1.045	**4.225**	**0.180**	**0.905**	0.962	0.979
Feature Depth	20	**0.099**	**0.694**	4.427	0.184	0.890	**0.963**	**0.982**

Table 4.1 Experiment 0: Results after training with the KITTI dataset. The different evaluation metrics are marked so that red marks metrics where lower is better and blue marks results where higher is better. The best result for each metric is marked in **bold**

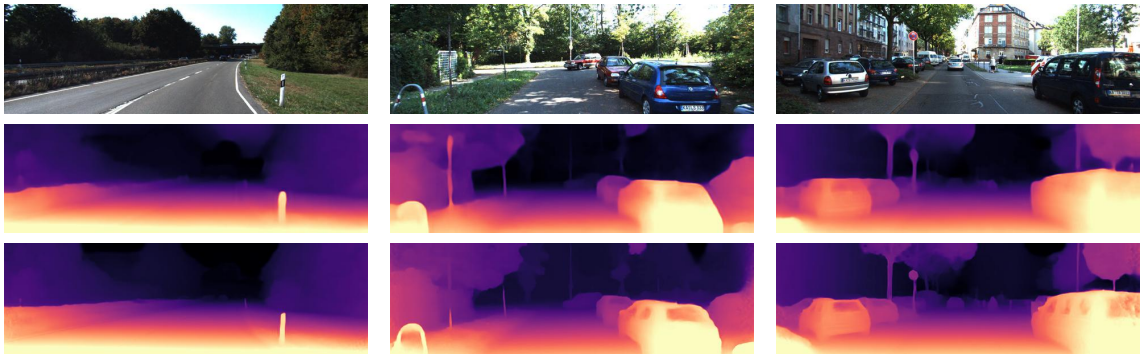


Figure 4.3: Experiment 0: Qualitative results on KITTI. First row: Original image, second row: Manydepth, third row: Feature Dept. Notice how Feature Depth does a better job for the sky in the left image, which is a low-textured area. Manydepth performs better on thin structures, and does a better job on the car windows in the center and right image

and evaluation metrics should develop during training.

4.4 Experiment 1: Training with Lyft

4.4.1 Setup

The Lyft dataset was extracted according to Section 3.6.2 with the shared data format in Section 3.6.1. This experiment utilized the 17 000 training frames and 600 validation frames from the front-facing camera, accompanied by a depth image containing a sparse set of depth values between 0.1 and 80 meters.

4.4.2 Results

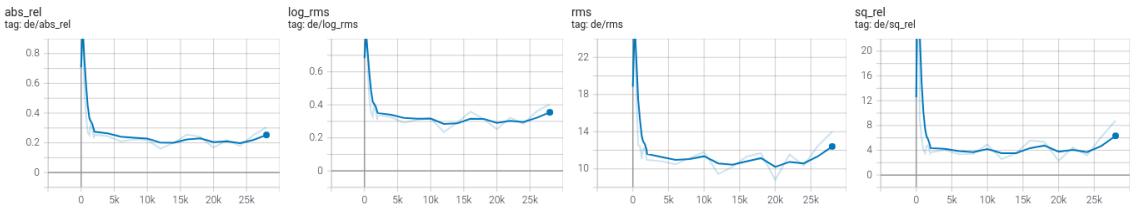
4.4.2.1 Manydepth

Training with Manydepth with Lyft showed great potential when looking at the qualitative results in Figure 4.6. The depth images have clear contrasts between segments, and the observed depth based on the color palette indicates a well-defined depth image. The best performing model was achieved after 20 000 iterations, translating to somewhere in epoch 14. This model achieved an absolute relative difference of 0.1797. The overall progression of the evaluation metrics and their final best score can be seen in Figure 4.4 and Table 4.2, respectively. The Manydepth model was able to run at 27 FPS on an NVIDIA T4 GPU running on the Oppdal cluster.

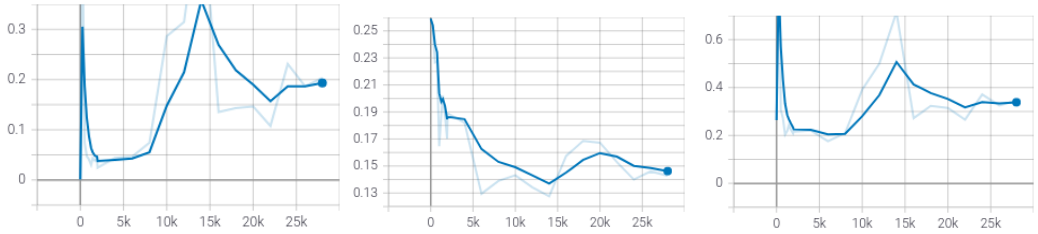
4.4.2.2 Feature Depth

Feature Depth also produced acceptable-looking depth maps for the Lyft data. However, the "punching hole" behavior is also visible here, possibly even more than in Manydepth.

The best performing model for Feature Depth was the model trained in epoch 28, achieving an absolute relative difference of 0.2300. The loss progressions and development of the evaluation



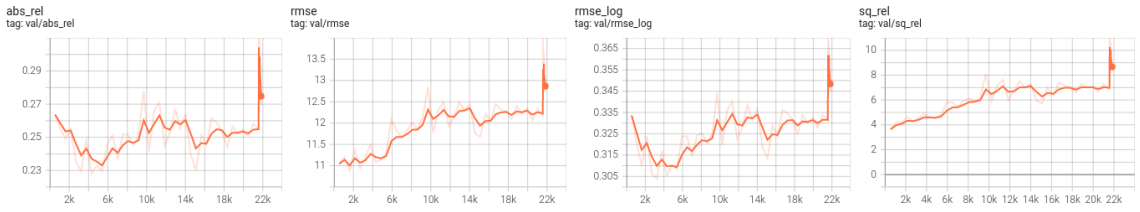
(a) Evaluation Metrics



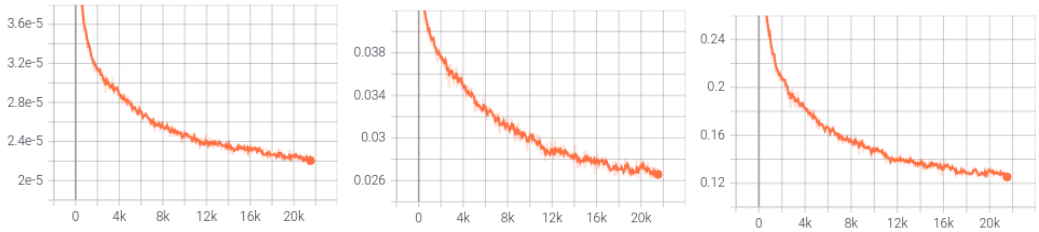
(b) Consistency Loss **(c)** Reprojection Loss **(d)** Total Loss

Figure 4.4: Experiment 1: Manydepth Lyft plots. Results after training Manydepth for 20 epochs with the Lyft dataset

metrics can be seen in Figure 4.5. Scores for all the evaluation metrics can be seen in Table 4.2.



(a) Evaluation Metrics



(b) Feature-metric Loss **(c)** Reconstruction Loss **(d)** Total Loss

Figure 4.5: Experiment 1: Feature Depth Lyft plots. Results after training Feature Depth for 40 epochs with the Lyft dataset. The spikes seen in the final part of the evaluation metrics plot is a failed attempt at doing test-time refinement, and should be ignored.

4.4.3 Discussion

Manydepth and Feature Depth show great potential according to what can be expected from an SSL-based system. By comparing the results from Manydepth and Feature Depth in Figure 4.6,

Dataset Name	FPS	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Manydepth	27	**0.1797**	**3.204**	**10.35**	**0.2848**	**0.7924**	**0.9201**	**0.9579**
Feature Depth	20	0.2300	5.714	11.67	0.3134	0.7735	0.8985	0.9402

Table 4.2 Experiment 1: Results after training with the Lyft dataset. Manydepth outperforms Feature Depth on all the evaluation metrics.

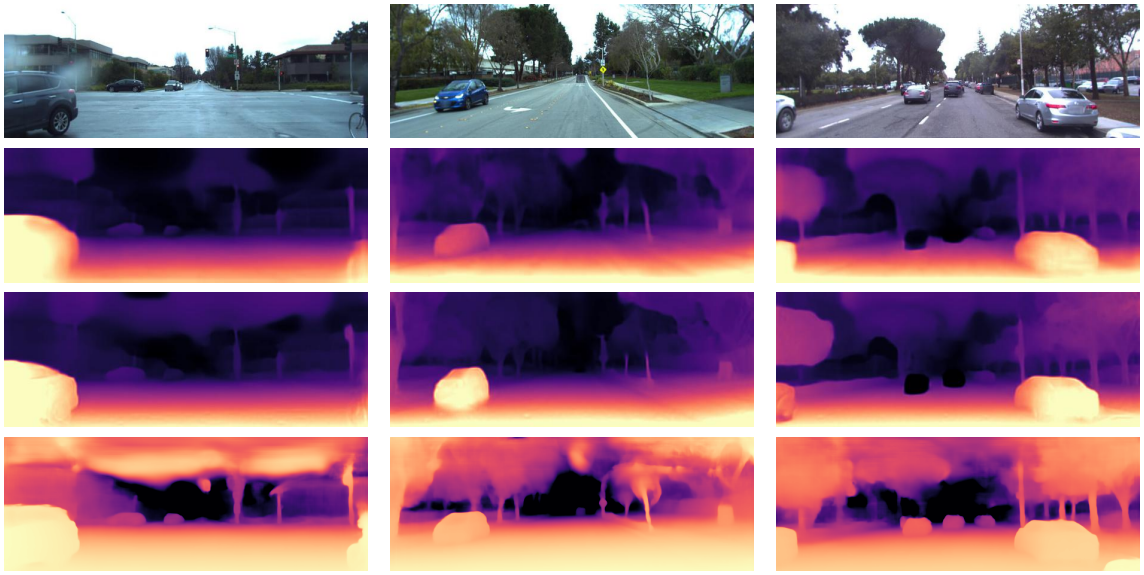


Figure 4.6: Experiment 1: Qualitative results on Lyft. First row: Original image, second row: Manydepth, third row: Feature Depth, fourth row: DPT. Notice that DPT do not have any problems with the "punching hole" effect, which is seen in both Manydepth and Feature Depth in the third image.

looking at the second and third column, one can see where Manydepth is superior to Feature Depth when looking at the car to the left in both images, where Feature Depth overestimates the proximity in the second image and underestimates it in the third. Looking at the results from DPT, it seems that it can identify relative depths correct with surprising accuracy when considering that no fine-tuning has been done with Lyft data. It is also worth mentioning that the color palette for DPT is slightly off compared to the others due to scaling differences.

A significant weakness of not using any supervision signal is visible in the third image, where both Manydepth and Feature Depth fails to predict a realistic depth value for the two cars in motion in the rightmost image. This shortcoming will be discussed further in Section 5.2

4.5 Experiment 2: Training with DDAD

4.5.1 Setup

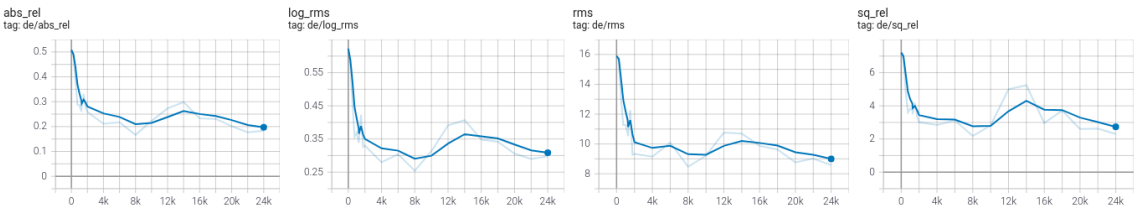
The DDAD dataset was extracted accordion to Section 3.6.3 with the shared data format in Section 3.6.1. This experiment utilized the 12 650 training frames and 900 validation frames from the front-facing camera, accompanied by a depth image generated from a long-range

lidar. The depth images contained depth values for up to 250 meters, but to keep the consistency with the Lyft dataset, all depth values over 80 meters were removed, leaving the final depth map having values between 0.1 and 80.0 meters.

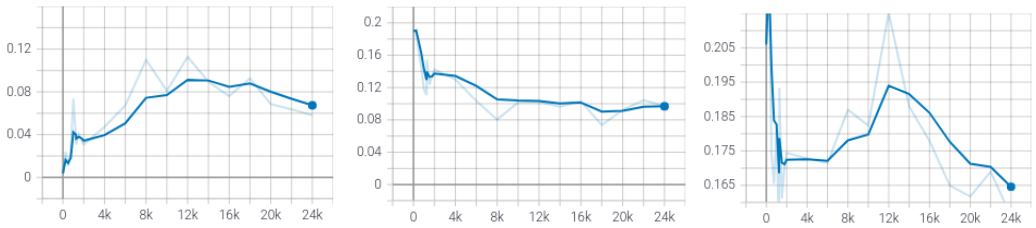
4.5.2 Results

4.5.2.1 Manydepth

Manydepth was trained for 24 000 training iterations divided into 20 epochs, with the best-performing model appearing in epoch 20. This indicates that training the model might not have converged yet, and could be further improved with more training. Nevertheless, this model generated an absolute relative difference of 0.1768 and could process 27 FPS on an NVIDIA T4 GPU. The FPS is comparable to the Lyft dataset due to the input size being equal for both datasets. The progression of the loss and evaluation metrics can be seen in Figure 4.7. The evaluation metrics from the best performing epoch and some qualitative results can be seen in Table 4.3 and Figure 4.9, respectively.



(a) Evaluation Metrics



(b) Consistency Loss **(c)** Reprojection Loss **(d)** Total Loss

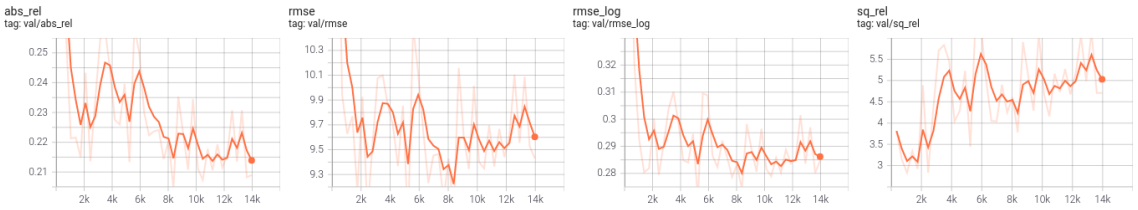
Figure 4.7: Experiment 2: Manydepth DDAD results. Results after training Manydepth for 20 epochs with the DDAD dataset

4.5.2.2 Feature Depth

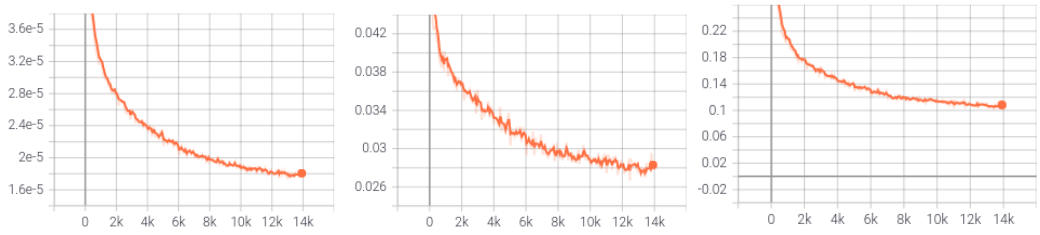
Feature Depth trained for 40 epochs, with the best performing epoch being number 28, which achieved an absolute relative difference score of 0.2139 and ran with 20 FPS on an NVIDIA T4 GPU, with an input size of 608 × 224.

4.5.3 Discussion

The qualitative results for Manydepth in Figure 4.9 show another weakness with SSL-based methods, as the depth images are a bit blurry. While performing training with the KITTI dataset,



(a) Evaluation metrics



(b) Feature-metric Loss **(c)** Reconstruction Loss **(d)** Total Loss

Figure 4.8: Experiment 2: Feature Depth DDAD plots. Results after training Feature Depth for 40 epochs on the DDAD dataset

Dataset Name	FPS	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Manydepth	27	0.1768	**2.623**	**9.024**	0.2897	0.7976	0.8880	0.9533
Feature Depth	20	0.2139	4.705	9.4300	0.2847	0.7583	**0.9106**	0.**9561**
PackNet* [30]	-	**0.162**	3.917	13.452	**0.269**	**0.823**	-	-

Table 4.3 Experiment 2: Results after training with the DDAD dataset. Results from the PackNet [30] paper is included, due to PackNet packnet being the main contribution in the paper where DDAD was introduced. Manydepth performs a lot better on the squared relative difference and RMSE metrics.

it was noticed that this behavior was present in the earlier epochs of training. During training, the predictions from the consistency model are also available. The blurriness was also noticeable in the depth model compared to the consistency model, which only uses a single input frame. Therefore, it is highly probable that the blurry depth images are due to a low number of available training frames. Experiment 3 will look more into a possible way to increase a dataset. Even though the depth images are blurrier, Manydepth produces more accurate segmentations. This is especially noticeable for the car in the middle column image and a more realistic looking depth estimation for the car on the left side in the third column image.

Purely looking at the qualitative results in Figure 4.9, Feature Depth performed better than Manydepth but achieved lower overall evaluation metrics. This behavior is mainly due to Feature Depth only using a single frame as input during testing, while Manydepth uses multiple. In addition, it has been observed that the monocular depth networks converge earlier than the depth network in Manydepth when comparing depth images during training of both Lyft and KITTI. This finding supports this hypothesis.

DPT seems to consistently misjudge the depth for the sky, which should be close to the max depth. This behavior is not a significant problem in itself but is worth noting.

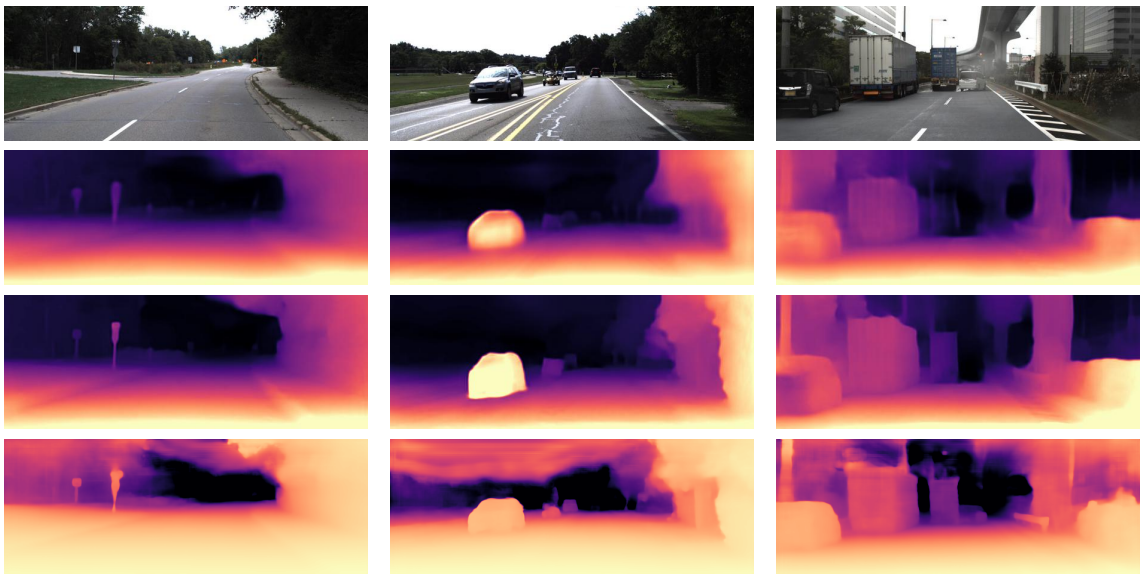


Figure 4.9: Experiment 2: Qualitative results on DDAD. First row: Original image, second row: Manydepth, third row: Feature Depth, fourth row: DPT. Notice that DPT do not have any problems with the "punching hole" effect, which is seen in both Manydepth and Feature Depth in the third image.

4.6 Experiment 3: Using Data from the Backward-facing Camera

4.6.1 Setup

The best way to improve any ML method is to introduce them to more data, with SSL-based methods being especially susceptible to this because they have no labels to supervise their development. Lyft and DDAD have a problem because they are relatively small datasets, especially compared to KITTI with 39 810 and Cityscapes with 69 731 training frames. Performing augmentations on the training frames in Lyft and DDAD will increase the performance, but only to a certain level.

Both Lyft and DDAD have camera data available from other cameras pointing in other directions. Unfortunately, using data from other camera angles simultaneously will confuse the SSL method's pose model, as this model learning is based on having the relative movement between the future and past frame being similar for the entire dataset. However, in theory, one could use data from the camera pointing directly backward, as the relative movement between frames is equal but in reverse temporal order compared to the front-facing camera. If this camera can be used, the dataset is effectively doubled in size.

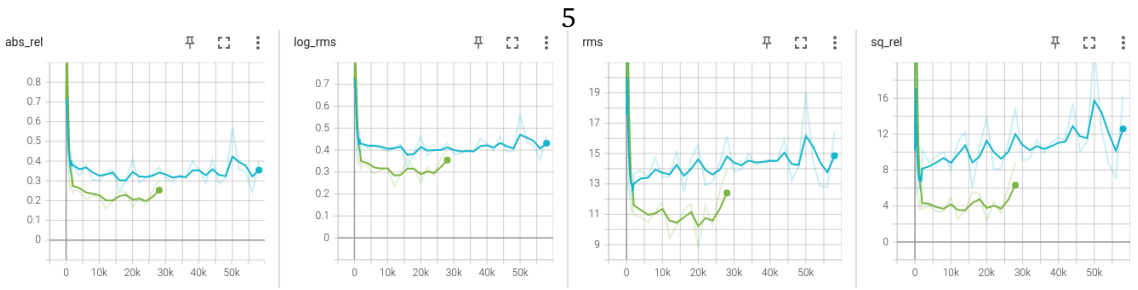
Testing this hypothesis consisted of first extracting data from the backward-facing camera in the Lyft dataset according to Section 3.6.1, only using the backward-facing camera. The relative index j was reversed, leaving a scene where the camera moved similarly to the forward-facing camera. This resulted in doubling the effective dataset to about 34 000 training frames.

4.6.2 Results

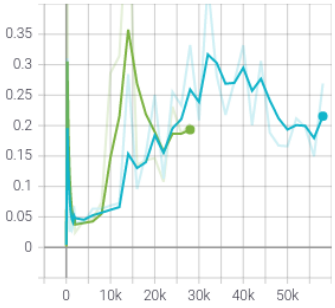
The results from this experiment resulted in a model performing worse than the model with only the forward-facing camera. The progression between the two models is shown in Figure 4.10. The final results of the model's best-performing epoch can be seen in Table 4.4.

4.6.3 Discussion

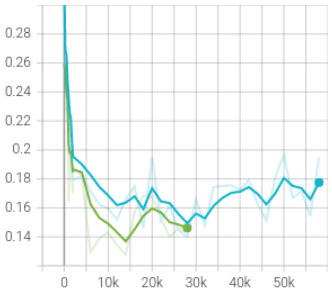
This result is unexpected as the relative movement between the frames should in-theory be similar. Another problem with this result is that the "punching hole" behavior mentioned in Experiment 1 and 2 is even more prominent for this model, which can be seen in Figure 4.11. This behavior might be increased due to the backward-facing camera seeing more moving objects and is discussed in more detail in Section 5.2



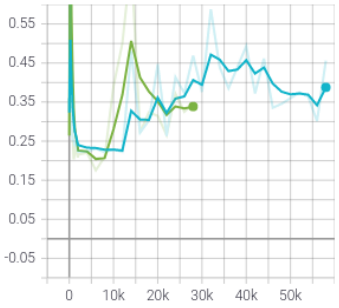
(a) Evaluation metrics



(b) Consistency Loss



(c) Reprojection Loss



(d) Total Loss

Figure 4.10: Experiment 3: Plots of using both datasets. Data is from the Front and Front+Back datasets, respectivly

Dataset Name	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Front only	**0.1797**	**3.204**	**10.35**	**0.2848**	0.7924	**0.9201**	**0.9579**
Front+Back	0.2418	7.189	12.78	0.3559	**0.8014**	0.9014	0.9345

Table 4.4 Experiment 3: Results using forward only and both backward and forward data. Using only the front-facing camera gives significantly better results than using both the front and backward-facing camera simultaneously. Both results is from training the Manydepth model, and the "Front-only" result is from Experiment 1.

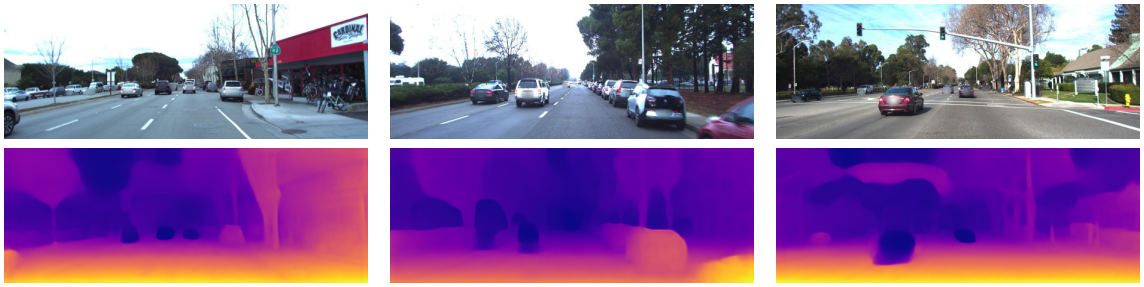


Figure 4.11: Experiment 3: Qualitative results on the combined Front+Back dataset. Notice how the "punching holes" covers the object worse than when the method is not trained with the backward-facing camera. Another noteworthy observation is that the depth image itself has a lighter color palette. This indicates that the scaling learned by Manydepth is less correct as well.

4.7 Experiment 4: Using NAP-lab Data

4.7.1 Setup

Data from the NAP-lab vehicle became available at a late stage in this thesis. This data contained video segments of the NAP-lab vehicle driving in Trondheim. Lidar data were also recorded during the runs, but due to this data not being available at an earlier point, time constraints resulted in this data not being extracted and used to calculate the evaluation metrics for this data. If this data becomes available later, a simple modification of the dataloaders should enable running an evaluation for this model. The data were extracted according to Section 3.6.1 and consisted of a total of 35 000 training frames and 1800 validation frames, all being extracted from the available videos.

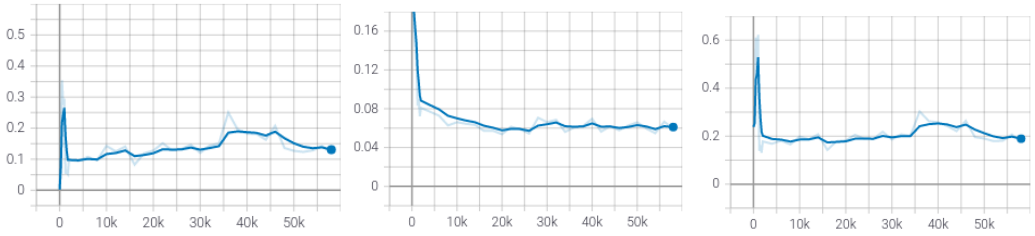
The overall best performing method in experiments one and two was Manydepth. Therefore, this method will be used when training with the NAP-lab data. The results will also show data from the DPT-hybrid model, which is fine-tuned on the KITTI dataset by Ranftl *et al.*

4.7.2 Results

Manydepth was trained for a total of 58 000 iterations over 20 epochs. Due to no depth maps being available, it is hard to say which model performed the best. Therefore, the model with the estimated lowest total loss is used for generating the qualitative results, which is the model trained in epoch 20. The qualitative results can be seen in Figure 4.13. The loss plots are shown in Figure 4.12.

4.7.3 Discussion

The qualitative results after training on only NAP-lab data show the strength of SSL-based solutions, as this dataset could not have been used in any supervised methods due to lacking the ground truth depth data. This dataset contained fewer frames of moving vehicles, which might be the reason for the lack of the "punching hole" anomalies seen for Lyft and DDAD.



(a) Consistency Loss **(b)** Reprojection Loss **(c)** Total Loss

Figure 4.12: Experiment 4: Manydepth NAP-lab plots. Results after training Manydepth for 20 epochs on the NAP-lab dataset. Notice how the loss values are more stable than Lyft and DDAD's loss progsession. No plots for the evaluation metrics are available, due to the NAP-lab dataset not containing data from any proximity sensors.

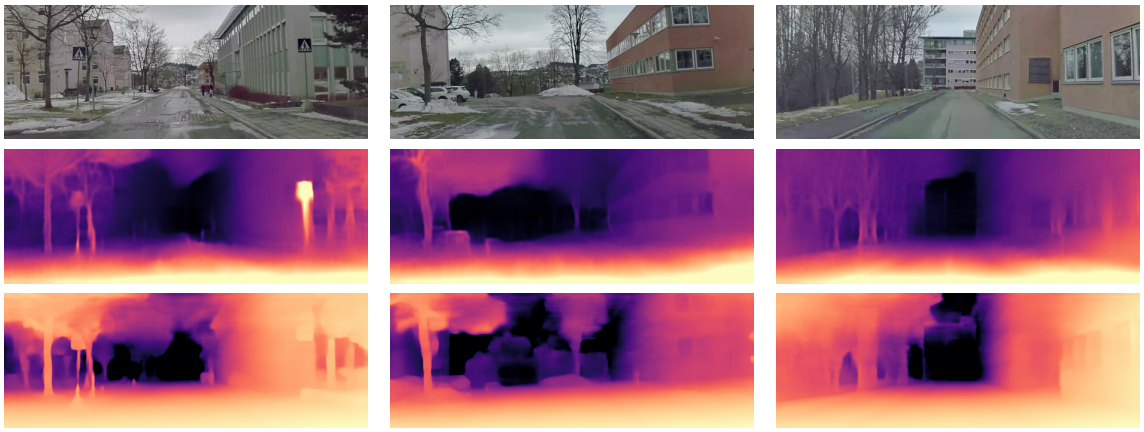


Figure 4.13: Experiment 4: Qualitative results on the NAP-lab dataset. First row: Original image, second row: Manydepth, third row: DPT. Both Manydepth and DPT have some problems with predicting depth values for the low textured areas, like the sky. The feature-metric loss from Feature Depth [46] could potentially improve the performance for these kinds of areas.

Chapter 5

Discussion

This chapter further explores the results obtained in the experiments. The chapter starts by comparing the trained models on the datasets that they were not trained on and comparing them. Some shortcomings and potential sources of error were introduced in the discussion sections in the experiments. These are further explained and reflected on in the next section of this chapter. Following is an analysis of the measures needed to start utilizing this system in an autonomous driving setting, together with a reflection of the supervised method's performance compared to the self-supervised ones. As mentioned in the methodology chapter, DPT can also produce semantic segmentations. This functionality is demonstrated before the chapter is concluded by answering the proposed research questions.

5.1 Difference in Results between Datasets

After having trained on three different datasets, it is interesting to see how well they perform on an unseen dataset. Manydepth was the model that performed the best for both datasets with a possibility to perform an evaluation. The Feature Depth model trained on Lyft was also included to see how well another method adapts to new data. The results from this comparison can be seen in Table 5.1.

It is a bit surprising to see how both the model trained on DDAD and NAP-lab do not adapt very well to the Lyft dataset, scoring very close to each other. What is surprising is that the model trained on the other dataset adapts a lot better to DDAD data, with the Manydepth model trained on Lyft outperforming the Feature Depth model trained on DDAD. One possible explanation could have been that DDAD contains fewer frames than Lyft and that more frames by themselves result in a better representation. However, this theory quickly falls when considering the 35 000 frames in the NAP-lab dataset, which performs worse than the DDAD model.

These results may indicate that the Lyft dataset contains an overall more representable set of scenes than DDAD and that this representation is more important than more frames by itself.

Method	Trained on	Evaluation Dataset	Abs Rel	Sq Rel	RMSE	RMSE log	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Manydepth	**Lyft**	**Lyft**	**0.1797**	**3.204**	**10.35**	**0.2848**	**0.7924**	**0.9201**	**0.9579**
Manydepth	DDAD	Lyft	*0.335*	6.993	13.909	0.391	0.543	0.804	0.910
Manydepth	NAP-lab	Lyft	0.336	5.930	13.886	0.461	0.478	0.756	0.881
Feature Depth	Lyft	Lyft	0.2300	*5.714*	*11.67*	*0.3134*	*0.7735*	*0.8985*	*0.9402*
Manydepth	Lyft	DDAD	*0.193*	**2.477**	*9.042*	**0.278**	0.720	*0.900*	**0.961**
Manydepth	**DDAD**	**DDAD**	**0.1768**	2.623	**9.024**	0.2897	**0.7976**	0.8880	0.9533
Manydepth	NAP-lab	DDAD	0.280	3.635	11.170	0.407	0.524	0.796	0.904
Feature Depth	DDAD	DDAD	0.2139	4.705	9.4300	*0.2847*	*0.7583*	**0.9106**	*0.9561*

Table 5.1 Comparing different models performance. The best metric for each dataset is shown in **bold** while the second best metric is shown in *italics*.

5.2 Noticable Shortcommings and Reflection

5.2.1 Punching Hole Behavior

The punching hole behavior was seen throughout the experiments. This effect shows itself when the depth model predicts depth values for a moving object, e.g., for cars, and is a known problem with SSL-based solutions. One observation done during training these methods is that the problem mainly manifests itself for objects moving in the same direction as the ego-vehicle.

An illustration is shown in Figure 5.1 from the DDAD dataset, where the depth images are generated by DPT and Manydepth trained on DDAD. Manydepth correctly predicts depth values for a person on a bike moving perpendicular to the vehicle. Another image shows correctly predicted depth for the cars in the opposite lane, moving in the opposite direction of the vehicle, while simultaneously failing to predict a correct depth value for the car moving on the right side.

The main reason why these holes appear is that the SSL-based dense depth methods, in most cases, assume a moving camera in a static scene, as stated in Section 2.6. Objects moving in the opposite direction to the ego-vehicle can be recognized as parked vehicles and stationary objects, which is common to see in all the datasets. Objects moving perpendicular to the scene do not have a changing depth value. However, the assumption is broken for objects moving in the same direction as the ego-vehicles and can vastly reduce the accuracy of the depth values.

5.2.1.1 Possible Solutions

Methods have been proposed to try to fix this problem. One known approach is to use auto-masking as Monodepth2 by Goudard *et al.*, which does not calculate the loss for the moving objects in the scene. Manydepth tries to combat this with the motion mask mentioned in Section 3.3. This mask looks at output from the consistency network, a standard monocular depth network, and the cost volume and identifies the moving objects as pixels where the cost volume and the consistency network disagree. Manydepth showed that this implementation helps a bit, but the problem is not solved, as shown in the experiment conducted in this thesis. More recent work by Li *et al.* [65] tries to solve this problem by predicting the constant relative background translation from ego-motion and the movement of the objects, all in a 3D translation

field. This field is used to achieve better motion predictions T, which again results in better-transformed views for the reconstruction part used in most SSL-based dense depth estimation tasks, described in Section 3.2.

All of the suggested improvements are based on finding the moving objects and dealing with them. A potential new way of doing this for future works is by utilizing the power of transformers, specifically vision transformers, in a self-supervised manner to detect the moving objects. The work done by Caron *et al.* [66] uses the data from the self-attention blocks in the visual transformer. As mentioned in Section 2.2, a transformer's attention can be compared to which part of the data a human agent would focus on. In this autonomous driving setting, we know humans focus on other moving objects, as other vehicles and pedestrians, when they are proximate to their vehicle. Using the work by Caron *et al.* could potentially improve this masking behavior and better address the problem of punching holes.

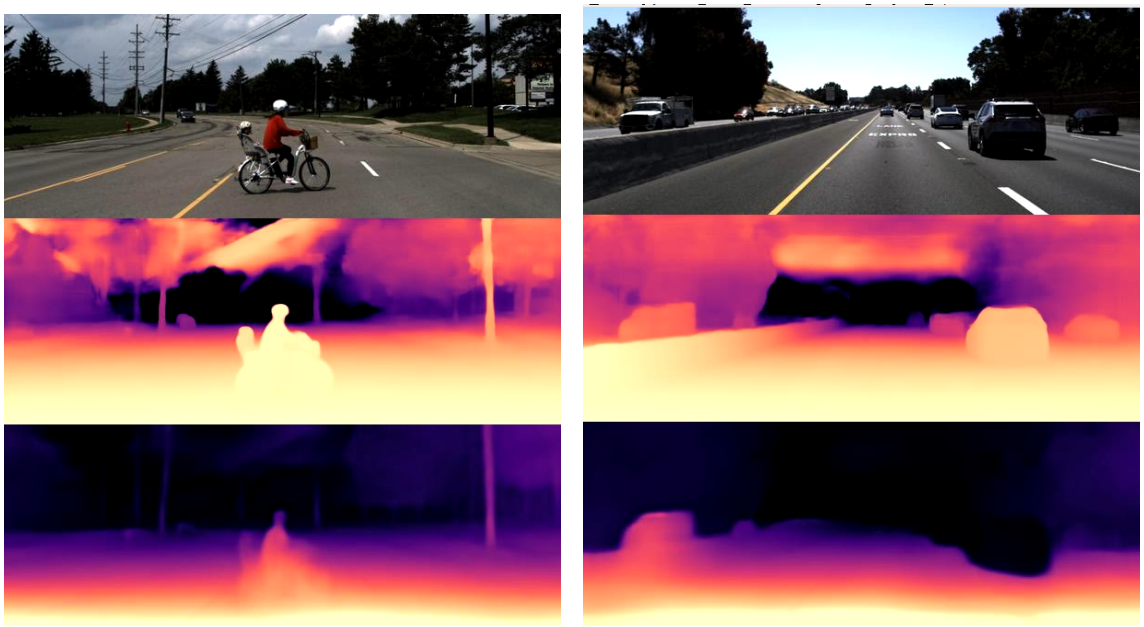


Figure 5.1: Punching hole behavior. Second row: DPT, third row: Manydepth. Here, a correct depth value is predicted for the objects moving in the opposite and perpendicular direction to the ego-vehicle, but fails for the object moving in the same direction.

5.2.2 Insignificant Amount of Data

The total number of training and validation frames for both the DDAD and Lyft datasets could preferably have been a lot higher. DDAD and Lyft had around 12 000 and 17 000 available training frames, far less than the benchmarking datasets KITTI and Cityscapes. Experiment 4 observed that simply using more frames is not necessarily an adequate solution to the problem in itself.

It is worth noting that a critical difference between the Lyft and DDAD dataset compared to the NAP-lab dataset is that Lyft, and especially the DDAD dataset, is handpicked from an enor-

mous amount of available data to represent many possible different situations an autonomous vehicle can experience. On the other hand, the NAP-lab dataset mostly contained videos of slow driving on the Gløshaugen campus, with a few videos from actual traffic.

5.2.2.1 Possible Solution

One possible way to improve the amount of available data, which was decided not to do in this thesis, is to use synthetical data, either from a simulator or simulated data with an extra layer of photorealism. Section 3.1.3.3 mentioned the work of Richter *et al.* [54], which uses game footage, sends it through a GAN, and outputs a photorealistic version of this footage with surprisingly convincing results. This method has been trained to simulate the frames in the Cityscapes dataset, which, as mentioned previously, is one of the most used benchmarking datasets used for autonomous vehicle-related tasks. Looking into the use of these kinds of photorealistic data could potentially vastly improve or complement existing datasets.

5.3 Usability in a an Autonomous Driving Setting

The methods tested in the experiments showed great potential for use in an autonomous driving setting. Although, the SSL-based methods have some critical problems that will need to be addressed before replacing sensors directly with ML. However, due to the produced point cloud from a range sensor like lidars or radars being moderately sparse, dense depth detections can aid and fill in the missing spots for a range sensor. In addition, the pose data from the SSL-based method's pose model can be utilized as an additional input to a state estimation system, e.g., as an addition to a Kalman filter.

5.3.1 Modifications needed to Operate in a Real-time Environment

An essential requirement for any system operating in a real-time environment like an autonomous driving setting, the system is required to operate in real-time. The currently best performing SSL-based method, Manydepth, operates at approximately 27 FPS on an NVIDIA T4 GPU. However, due to the T4 GPU being significantly more potent than the hardware found in an autonomous platform, the inference times reported in the experiments are not representable for running on hardware in an autonomous platform. In addition, the input size used when generating the depth images in the experiments could also have been increased to introduce more detail to the depth images, decreasing the expected FPS even more, probably into the 10s, maybe even lower.

One possible improvement is to optimize the model with an optimizing tool like TensorRT instead of using native PyTorch when performing inference. Using TensorRT in a lower precision mode like FP16 or even INT8 could potentially double the inference time, leaving more room to increase the image size and still get the necessary inference speeds needed to operate in a real-time environment.

5.3.2 Shared Base Model

There exist multiple approaches to achieve self-driving, where there are two main categories. The more traditional approach uses a pipeline structure containing separate, modular systems for perception, state estimation, mapping, path planning, and control. The second and more modern approach uses an end-to-end structure, where a decision-taking entity, like a neural network, uses the raw data from the sensor suite on the vehicle and outputs actions like steering and throttle outputs. Other more specialized solutions exist as well [67]. Both of these solutions offer strengths and weaknesses, with, e.g., a pipeline-based structure being more straightforward to debug than an end-to-end solution, but can simultaneously consist of too many tunable parameters, increasing the possibility for human error, which is not a case for end-to-end based solutions.

A possibility here for a common middle ground, achieving the best-of-both-worlds is using architectures like the DPT, where in theory it would be possible to modify the DPT architecture to be a multi-headed architecture, where each network could be fine-tuned to its specific task. Having a shared base architecture like this can streamline training while still being relatively easy to debug, as one can view the output from each task-specific network. This structure resembles how Tesla creates their AutoPilot system with its "HydraNet" [68]. Their specific network architecture is unknown, but it would be interesting to test a transformer-based architecture using this approach.

5.4 Comparing Supervised and Self-supervised

The SSL-based systems still showed their strengths by generating a depths map with no other supervision signal than the data itself. For some selected cases, like poles and signs in the first image in Figure 4.9, the SSL-based methods perform better than DPT. Even though this specific case is not that useful in itself, it still shows that SSL-based methods should not be ignored.

Throughout the experiments, DPT was, by far, the method that generated the best looking qualitative results for all the datasets, even though no fine-tuning was done with the datasets used. The ability to perform at this level without any fine-tuning indicates that the DPT system genuinely can generalize to an ill-posed problem of this magnitude better than many other existing solutions.

DPT is by far one of the currently more innovative approaches to dense depth estimation. As one is now starting to see some of the potentials of using transformers for visual tasks, much like using transformers for NLP tasks skyrocketed the performance and NLP models, it is truly inspiring to see how far these solutions can go.

However, DPT is still a supervised method, requiring dense predictions during training and can use sparse data, like depth maps generated from lidar scans, when fine-tuning on a separate dataset. While this is not a problem in itself, it would be beneficial if this model could be trained unsupervised. Unfortunately, no existing solutions have yet investigated this problem. However, by building upon the works of Caron *et al.* [66], it is most likely only a question of time, as vision transformers and especially DPT still are relatively recent works, being published in October 2020 and March 2021, respectively.

5.4.1 DPT's Semantic Segmentation

Figure 5.2: Semantic Segmentation on NAP-lab Data. DPT is able to produce believable segmentation maps for an use-case where it has seen very little examples.

Section 3.1.2.2 mentioned that DPT can perform multiple dense prediction tasks. One of the other tasks it is trained on is semantic segmentation. While semantic segmentation is not a part of this thesis, it is still interesting to see how well it can perform on unseen data. As mentioned earlier, the DPT-Hybrid-KITTI model used to generate the depth images in this thesis is fine-tuned on the KITTI dataset. The central part of the fine-tuning is the actual training of the task-specific head network. However, due to a lack of semantic segmentation data for KITTI, this pretraining is not done for the semantic segmentation head network. The ADE20K [69] dataset is used here instead, which contains some semantic maps of relevant objects, such

as cars, roads, and trucks. It still contains significantly fewer examples than, e.g., Cityscapes, but surprisingly enough, it can still perform decently well on the segmentation task, which builds upon the statement of DPT's ability to generalize. The segmented images can be seen in Figure 5.2

5.5 Proposing a Novel Architecture

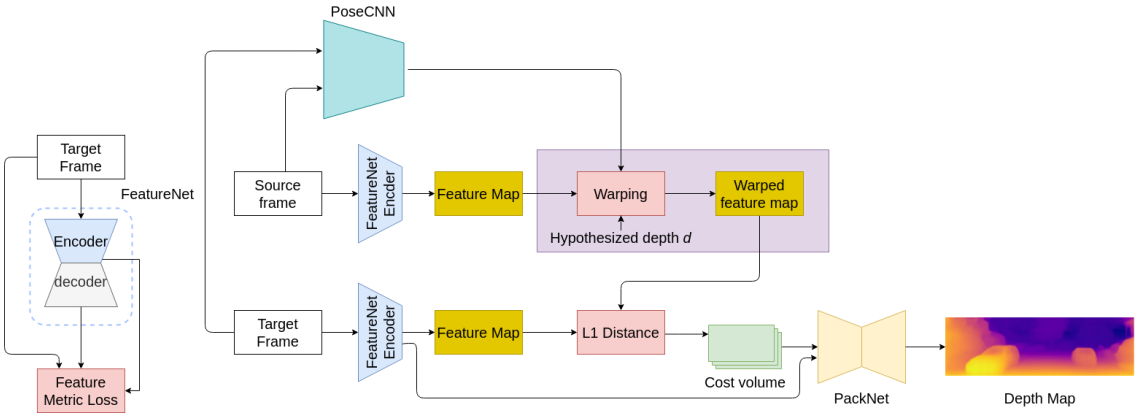


Figure 5.3: Novel Architecture.

Based on the findings in this thesis, a novel architecture that combines the best contributions from the current SotA methods. This thesis has already explored the feature-metric loss in Feature Depth and the cost-volume-based method of using multiple images in the depth network, which both are methods performing at SotA levels. Another SotA method this thesis considered exploring is PackNet [30], with its novel architecture based on 3D convolutions that essentially learn to compress and decompress an image without losing details.

One notable observation for the three SotA methods is that they all contribute to different parts of the fundamental architecture of self-supervised depth estimation. Therefore, after looking in-depth at Feature Depth and Manydepth, this thesis proposes combining the different contributions into a single architecture that can benefit from all the previous contributions.

The proposed architecture can be seen in Figure 5.3. The architecture starts with a target image and multiple source images, equally as Manydepth. These are fed into a feature encoder, trained by FeatureNet from the Feature Depth architecture, and the images are transformed into feature maps. The images are also fed into a pose model θ_{pose} that estimates the pose between the source and target images. With this pose, the feature maps from the source image are warped into the target image's pose. Here, the L1 distance between the target and warped features are fed into a cost volume for each hypothesized depth d and is repeated for each depth plane $d \in \mathcal{P}$. Together with the features from the target frames, this cost volume is all fed into the PackNet depth network from the PackNet paper, which utilizes 3D convolutions to save finer details in the image better and produce a depth map target frame.

The idea and motivation behind this architecture are that the contribution of cost volumes will allow the architecture to use the temporal data often found in a sequence of images. The feature-metric loss will help the network in low-textured areas like the sky, flat surfaces, and low-lit areas. Using the PackNet architecture will enable the network to preserve better the more delicate details found in the features and ensure a sharper and more precise depth image.

5.6 Fulfillment of the Research Questions

RQ 1: The first research question asked if self-supervised-based dense depth estimation methods can achieve the same performance as supervised methods. The experiments have shown that DPT, in most cases, is superior to the SSL-based methods. However, the main downside with supervised methods and DPT is that they require high-quality depth data generated by expensive range sensors like lidars and radars. This might not be the case when high-quality depth data is available, like for some UAVs, indoor environments, or even in medical applications. Being a transformer-based system, DPT only starts performing at the exceptional level seen in the experiments after a thorough pretraining. If the domain at hand should be too different from a known environment, the self-supervised methods have a great chance of performing better than supervised methods.

Thus, the answer to the research question is: For most cases, no, but self-supervised methods have an advantage in domains where there is little to no available depth data from high-quality depth sensors.

RQ 2: The second research question sought to analyze whether or not dense depth estimation methods could replace high-detailed depth sensors like lidars. Self-supervised methods have a great potential for understanding depth without any supervision signals when looking at the results achieved in the experiments. However, the self-supervised-based methods have a critical flaw with their "punching-hole" behavior when the assumption of a static scene is broken with moving objects like other cars. This flaw alone excludes self-supervised methods as a potential candidate to replace a range sensor, as reliably detecting other moving vehicles is a fundamental requirement for any range detector.

Supervised methods like DPT are not exposed to this problem as they can utilize depth information as supervision signals during training. However, a model cannot have 100% accuracy, and one can even extend this to say that not any range sensor can have 100% accuracy. Nevertheless, as detecting proximity to other objects is a critical part of most tasks where they are present in the first place, an image-based method is too exposed to inaccuracy. These inaccuracies may come from camera anomalies like blurriness and low contrast or environmental ambiguities like direct sunlight, darkness, and fog, replacing high-detailed depth sensors like lidars or radars with an image-based dense depth estimation system not be reliable, as one introduces an additional point of failure after sensor failure itself.

However, a point worth noting is that dense depth estimation tasks are highly effective in filling out the missing information from the sparse point clouds generated by depth sensors. Thus

using a combination of high-detailed proximity sensors and dense depth estimation methods can improve the overall depth information. Self-supervised methods can also compliment state estimation systems with their pose estimations.

RQ 3: After looking at the performance of the self-supervised method compared to the supervised methods, the final research question investigated which measures can be done to improve the self-supervised methods. The experiments observed that the recently published state-of-the-art methods already had taken some measures. One measure is to utilize the temporal information found in sequences available at test time. Manydepth is not the first to do this, but it is the first model to do it smartly. Other improvements consist of using losses tailored to improve areas where current state-of-the-art methods fail. The Feature Depth method does this with its feature-metric loss, which improves the loss calculations in low-textured areas.

Currently, the most critical flaw with self-supervised dense depth estimation methods is the "punching hole" behavior. Some possible solutions to this problem have already been introduced, but the problem itself is hard to fix because the assumption of a static scene is close to impossible, e.g., an autonomous driving setting.

This thesis has also presented a novel measure that can be taken to improve the field, specifically a novel architecture using the combined contributions from the current state-of-the-art in a way that aids the different methods in places where they have shortcomings. Therefore, this thesis suggests that the most significant measure that can be taken to improve the existing state-of-the-art self-supervised methods is by implementing the novel architecture.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The primary goal of this thesis was to gain the knowledge needed to suggest a novel architecture for a SSL depth estimation system that would have the potential to improve the current SotA performance. This system should also be usable in an autonomous driving-setting, by being able to run in real-time. As novel contributions are being published continuously, this is no easy task. Both Manydepth and DPT were published two months into writing this thesis, and both of them can be considered the current SotA for SSL and supervised depth estimation. However, it should be easier to adapt a potential novel architecture with new contributions by gaining fundamentally solid knowledge in the field of SSL depth estimation.

Gaining this knowledge was done through first building some basic knowledge into deep learning and depth estimation. Following was a study of essential works in supervised and SSL depth estimation before looking into the current SotA. After choosing two SSL methods, these were studied in-depth and trained on new data. A combination of theory and practice showed how these method's contributions aided the field of SSL depth estimation and thus substantiated the choices made for the novel architecture.

There has been considerable research into different ways of improving the field of SSL depth estimation. Nevertheless, during this thesis, it was observed that these contributions often are contributions that focus on a single element in the architecture. Still, there exist very few cases of these approaches being combined on a larger scale. Some contributions take inspiration from each other and implement smaller contributions from other papers, e.g., using minimal photometric loss rather than average. Still, there is untouched potential in combining the more significant findings into a completely novel architecture.

If this thesis could be redone, it would probably have also included a deep dive into the PackNet architecture, with an associated experiment consisting of training the PackNet architecture with the Lyft and DDAD datasets. With this, one would have all the information needed to validate if PackNet should be included in the novel architecture. Due to both time constraints and limitations in how much work can be done in a single thesis, there was no time to implement the actual architecture itself. However, the actual implementation of this architecture could be a potential starting point for a new thesis.

6.2 Future Work

This section will suggest some potentials for theses and works that can continue the work done and use the results found in this thesis.

6.2.1 Implementing the Novel Architecture

A natural point of continuation would be to implement the proposed novel architecture. This job would mainly consist of merging these contributions into a single architecture by utilizing the existing codebases. When one has a functional architecture, this needs to be trained and validated, preferably on the KITTI and Cityscapes dataset, following the norm in the field.

6.2.2 Semi-supervised Dense Depth Estimation

When having an autonomous driving testing platform available, one automatically gets some extra data points available for free. As mentioned in Section 5.3.2, an autonomous driving pipeline often consists of multiple modules, where these modules either use or produce data regarding the vehicle's state. With this, one often has information about the vehicle's position, ego-motion, and intended path. Due to the reprojection training depending on the relative pose between a source and a target image, as discussed in Section 3.2.2, this pose needs to be learned. However, if one were to use more sophisticated systems to predict the pose, one could potentially increase the performance of the depth estimator. PackNet has already looked into using the vehicle's velocity as an input during training. This extra data point increased the performance quite substantially, and it supports the claim that using external data points increases the performance.

6.2.3 Applying Dense Depth Estimation to a Real-world Use Case

"Statens vegvesen" is a governmental agency responsible for Norway's public roads, which includes planning, construction, and maintenance. They have an ongoing project where they try to detect and store info regarding road damage. They collect information about the roads from a camera mounted on a car, where the camera snaps an image every 10 meters and stores it together with a geographic position obtained from a GNSS receiver. Using SOTA object detection and semantic segmentation techniques, one can detect the road damage from the images. An optimal solution for this problem would be to have the exact latitude and longitude of the road damage. However, object detection methods only predict where the damage is in a 2D image and not in the 3D world.

A possible solution to this problem is to use dense depth estimation. For each image produced, one can generate a dense depth map. Together with a geographic position for the origin point of the camera capturing the image together with either a bounding box or segmentation containing the road damage, one could project the pixels containing the damage into a 3D projection, as described in Section 3.2.2.2, where each point represents a latitude, longitude, and elevation, essentially giving the exact geographic coordinate for the road damage.

Bibliography

[1] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines vinod nair," vol. 27, Jun. 2010, pp. 807–814.

[2] G. Sanderson. (2017). "What is backpropagation really doing? | deep learning, chapter 3," Youtube, [Online]. Available: https://www.youtube.com/watch?v=Ilg3gGewQ5U.

[3] G. Sanderson. (2017). "Backpropagation calculus | deep learning, chapter 4," Youtube, [Online]. Available: https://www.youtube.com/watch?v=tIeHLnjs5U8.

[4] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011, https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[5] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," *arXiv e-prints*, arXiv:1212.5701, arXiv:1212.5701, Dec. 2012. arXiv: 1212.5701 [cs.LG].

[6] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv e-prints*, arXiv:1412.6980, arXiv:1412.6980, Dec. 2014. arXiv: 1412.6980 [cs.LG].

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[8] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[9] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv e-prints*, arXiv:1409.1556, arXiv:1409.1556, Sep. 2014. arXiv: 1409.1556 [cs.CV].

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv e-prints*, arXiv:1512.03385, arXiv:1512.03385, Dec. 2015. arXiv: 1512.03385 [cs.CV].

[11] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *arXiv e-prints*, arXiv:1602.07261, arXiv:1602.07261, Feb. 2016. arXiv: 1602.07261 [cs.CV].

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv e-prints*, arXiv:1706.03762, arXiv:1706.03762, Jun. 2017. arXiv: 1706.03762 [cs.CL].

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv e-prints*, arXiv:1810.04805, arXiv:1810.04805, Oct. 2018. arXiv: 1810.04805 [cs.CL].

[14] A. Radford and I. Sutskever, "Improving language understanding by generative pre-training," in *arXiv*, 2018.

[15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. arXiv: 2005.14165 [cs.CL].

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv e-prints*, arXiv:2010.11929, arXiv:2010.11929, Oct. 2020. arXiv: 2010.11929 [cs.CV].

[18] L. Jing and Y. Tian, "Self-supervised Visual Feature Learning with Deep Neural Networks: A Survey," *arXiv e-prints*, arXiv:1902.06162, arXiv:1902.06162, Feb. 2019. arXiv: 1902.06162 [cs.CV].

[19] V. Keshav and F. Delattre, "Self-supervised visual feature learning with curriculum," *arXiv e-prints*, arXiv:2001.05634, arXiv:2001.05634, Jan. 2020. arXiv: 2001.05634 [cs.CV].

[20] S. Gidaris, P. Singh, and N. Komodakis, "Unsupervised Representation Learning by Predicting Image Rotations," *arXiv e-prints*, arXiv:1803.07728, arXiv:1803.07728, Mar. 2018. arXiv: 1803.07728 [cs.CV].

[21] Z. Feng, C. Xu, and D. Tao, "Self-supervised representation learning by rotation feature decoupling," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 10 356–10 366.

[22] M. Noroozi and P. Favaro, "Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles," *arXiv e-prints*, arXiv:1603.09246, arXiv:1603.09246, Mar. 2016. arXiv: 1603.09246 [cs.CV].

[23] G. Larsson, M. Maire, and G. Shakhnarovich, "Colorization as a Proxy Task for Visual Understanding," *arXiv e-prints*, arXiv:1703.04044, arXiv:1703.04044, Mar. 2017. arXiv: 1703.04044 [cs.CV].

[24] R. Zhang, P. Isola, and A. A. Efros, "Colorful Image Colorization," *arXiv e-prints*, arXiv:1603.08511, arXiv:1603.08511, Mar. 2016. arXiv: 1603.08511 [cs.CV].

[25] I. Misra, C. L. Zitnick, and M. Hebert, "Shuffle and Learn: Unsupervised Learning using Temporal Order Verification," *arXiv e-prints*, arXiv:1603.08561, arXiv:1603.08561, Mar. 2016. arXiv: 1603.08561 [cs.CV].

[26] C. Chen, B. Wang, C. Lu, N. Trigoni, and A. Markham, "A survey on deep learning for localization and mapping: Towards the age of spatial machine intelligence," Jun. 2020.

[27] R. Szeliski and Sing Bing Kang, "Shape ambiguities in structure from motion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 5, pp. 506–512, 1997. DOI: 10.1109/34.589211.

[28] C. Godard, O. Mac Aodha, and G. J. Brostow, "Unsupervised Monocular Depth Estimation with Left-Right Consistency," *arXiv e-prints*, arXiv:1609.03677, arXiv:1609.03677, Sep. 2016. arXiv: 1609.03677 [cs.CV].

[29] C. Godard, O. Mac Aodha, M. Firman, and G. Brostow, "Digging Into Self-Supervised Monocular Depth Estimation," *arXiv e-prints*, arXiv:1806.01260, arXiv:1806.01260, Jun. 2018. arXiv: 1806.01260 [cs.CV].

[30] V. Guizilini, R. Ambrus, S. Pillai, A. Raventos, and A. Gaidon, "3D Packing for Self-Supervised Monocular Depth Estimation," *arXiv e-prints*, arXiv:1905.02693, arXiv:1905.02693, May 2019. arXiv: 1905.02693 [cs.CV].

[31] Z. Zhang, J. Qiao, S. Lin, and H. Liu, "Weakly supervised monocular depth estimation method based on stereo matching labels," *Journal of Electronic Imaging*, vol. 29, Oct. 2020. DOI: 10.1117/1.JEI.29.5.053013.

[32] L. Madhuanand, F. Nex, and M. Y. Yang, "Self-supervised monocular depth estimation from oblique UAV videos," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 176, pp. 1–14, Jun. 2021. DOI: 10.1016/j.isprsjprs.2021.03.024. arXiv: 2012.10704 [cs.CV].

[33] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[34] P. R. Palafox, J. Betz, F. Nobis, K. Riedl, and M. Lienkamp, "Semanticdepth: Fusing semantic segmentation and monocular depth estimation for enabling autonomous driving in roads without lane lines," *Sensors*, vol. 19, no. 14, 2019, ISSN: 1424-8220. [Online]. Available: https://www.mdpi.com/1424-8220/19/14/3224.

[35] N. Zioulis, A. Karakottas, D. Zarpalas, and P. Daras, "OmniDepth: Dense Depth Estimation for Indoors Spherical Panoramas," *arXiv e-prints*, arXiv:1807.09620, arXiv:1807.09620, Jul. 2018. arXiv: 1807.09620 [cs.CV].

[36] L. Chen, W. Tang, N. W. John, T. Ruan Wan, and J. J. Zhang, "Augmented Reality for Depth Cues in Monocular Minimally Invasive Surgery," *arXiv e-prints*, arXiv:1703.01243, arXiv:1703.01243, Mar. 2017. arXiv: 1703.01243 [cs.CV].

[37] D. Eigen, C. Puhrsch, and R. Fergus, "Depth Map Prediction from a Single Image using a Multi-Scale Deep Network," *arXiv e-prints*, arXiv:1406.2283, arXiv:1406.2283, Jun. 2014. arXiv: 1406.2283 [cs.CV].

[38] I. Laina, C. Rupprecht, V. Belagiannis, F. Tombari, and N. Navab, "Deeper Depth Prediction with Fully Convolutional Residual Networks," *arXiv e-prints*, arXiv:1606.00373, arXiv:1606.00373, Jun. 2016. arXiv: 1606.00373 [cs.CV].

[39] R. Wang, S. M. Pizer, and J.-M. Frahm, "Recurrent Neural Network for (Un-)supervised Learning of Monocular VideoVisual Odometry and Depth," *arXiv e-prints*, arXiv:1904.07087, arXiv:1904.07087, Apr. 2019. arXiv: 1904.07087 [cs.CV].

[40] K. G. Lore, K. Reddy, M. Giering, and E. A. Bernal, "Generative adversarial networks for depth map estimation from rgb video," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2018, pp. 1258–12 588. DOI: 10.1109/CVPRW.2018.00163.

[41] S. Farooq Bhat, I. Alhashim, and P. Wonka, "AdaBins: Depth Estimation using Adaptive Bins," *arXiv e-prints*, arXiv:2011.14141, arXiv:2011.14141, Nov. 2020. arXiv: 2011.14141 [cs.CV].

[42] R. Ranftl, A. Bochkovskiy, and V. Koltun, "Vision Transformers for Dense Prediction," *arXiv e-prints*, arXiv:2103.13413, arXiv:2103.13413, Mar. 2021. arXiv: 2103.13413 [cs.CV].

[43] J. Watson, M. Firman, G. J. Brostow, and D. Turmukhambetov, "Self-supervised monocular depth hints," in *The International Conference on Computer Vision (ICCV)*, Oct. 2019.

[44] T. Zhou, M. Brown, N. Snavely, and D. G. Lowe, "Unsupervised Learning of Depth and Ego-Motion from Video," *arXiv e-prints*, arXiv:1704.07813, arXiv:1704.07813, Apr. 2017. arXiv: 1704.07813 [cs.CV].

[45] S. Vijayanarasimhan, S. Ricco, C. Schmid, R. Sukthankar, and K. Fragkiadaki, "SfM-Net: Learning of Structure and Motion from Video," *arXiv e-prints*, arXiv:1704.07804, arXiv:1704.07804, Apr. 2017. arXiv: 1704.07804 [cs.CV].

[46] C. Shu, K. Yu, Z. Duan, and K. Yang, "Feature-metric Loss for Self-supervised Learning of Depth and Egomotion," *arXiv e-prints*, arXiv:2007.10603, arXiv:2007.10603, Jul. 2020. arXiv: 2007.10603 [cs.CV].

[47] J. Watson, O. M. Aodha, V. Prisacariu, G. Brostow, and M. Firman, "The Temporal Opportunist: Self-Supervised Multi-Frame Monocular Depth," in *Computer Vision and Pattern Recognition (CVPR)*, 2021.

[48] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," *arXiv e-prints*, arXiv:1604.01685, arXiv:1604.01685, Apr. 2016. arXiv: 1604.01685 [cs.CV].

[49] R. Kesten, M. Usman, J. Houston, T. Pandya, K. Nadhamuni, A. Ferreira, M. Yuan, B. Low, A. Jain, P. Ondruska, S. Omari, S. Shah, A. Kulkarni, A. Kazakova, C. Tao, L. Platinsky, W. Jiang, and V. Shet, *Lyft level 5 perception dataset 2020*, https://level5.lyft.com/dataset/, 2019.

[50] *Waymo open dataset: An autonomous driving dataset*, 2019.

[51] D. S. Galteland, "Self-supervised learning - project thesis," Dec. 2020.

[52] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[53] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*, 2017. eprint: arXiv:1705.05065. [Online]. Available: https://arxiv.org/abs/1705.05065.

[54] S. R. Richter, H. A. AlHaija, and V. Koltun, "Enhancing photorealism enhancement," *arXiv:2105.04619*, 2021.

[55] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Second. Cambridge University Press, ISBN: 0521540518, 2004.

[56] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000. DOI: 10.1109/34.888718.

[57] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. DOI: 10.1109/TIP.2003.819861.

[58] P. Heise, S. Klose, B. Jensen, and A. Knoll, "Pm-huber: Patchmatch with huber regularization for stereo matching," in *2013 IEEE International Conference on Computer Vision*, 2013, pp. 2360–2367. DOI: 10.1109/ICCV.2013.293.

[59] V. Guizilini, R. Hou, J. Li, R. Ambrus, and A. Gaidon, "Semantically-Guided Representation Learning for Self-Supervised Monocular Depth," *arXiv e-prints*, arXiv:2002.12319, arXiv:2002.12319, Feb. 2020. arXiv: 2002.12319 [cs.CV].

[60] G. Lin, A. Milan, C. Shen, and I. Reid, "RefineNet: Multi-Path Refinement Networks for High-Resolution Semantic Segmentation," *arXiv e-prints*, arXiv:1611.06612, arXiv:1611.06612, Nov. 2016. arXiv: 1611.06612 [cs.CV].

[61] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer," *arXiv e-prints*, arXiv:1907.01341, arXiv:1907.01341, Jul. 2019. arXiv: 1907.01341 [cs.CV].

[62] J. Kannala and S. Brandt, "A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 8, pp. 1335–1340, 2006. DOI: 10.1109/TPAMI.2006.153.

[63] R. Garg, V. K. BG, G. Carneiro, and I. Reid, "Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue," *arXiv e-prints*, arXiv:1603.04992, arXiv:1603.04992, Mar. 2016. arXiv: 1603.04992 [cs.CV].

[64] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: 1912.05848 [cs.DC].

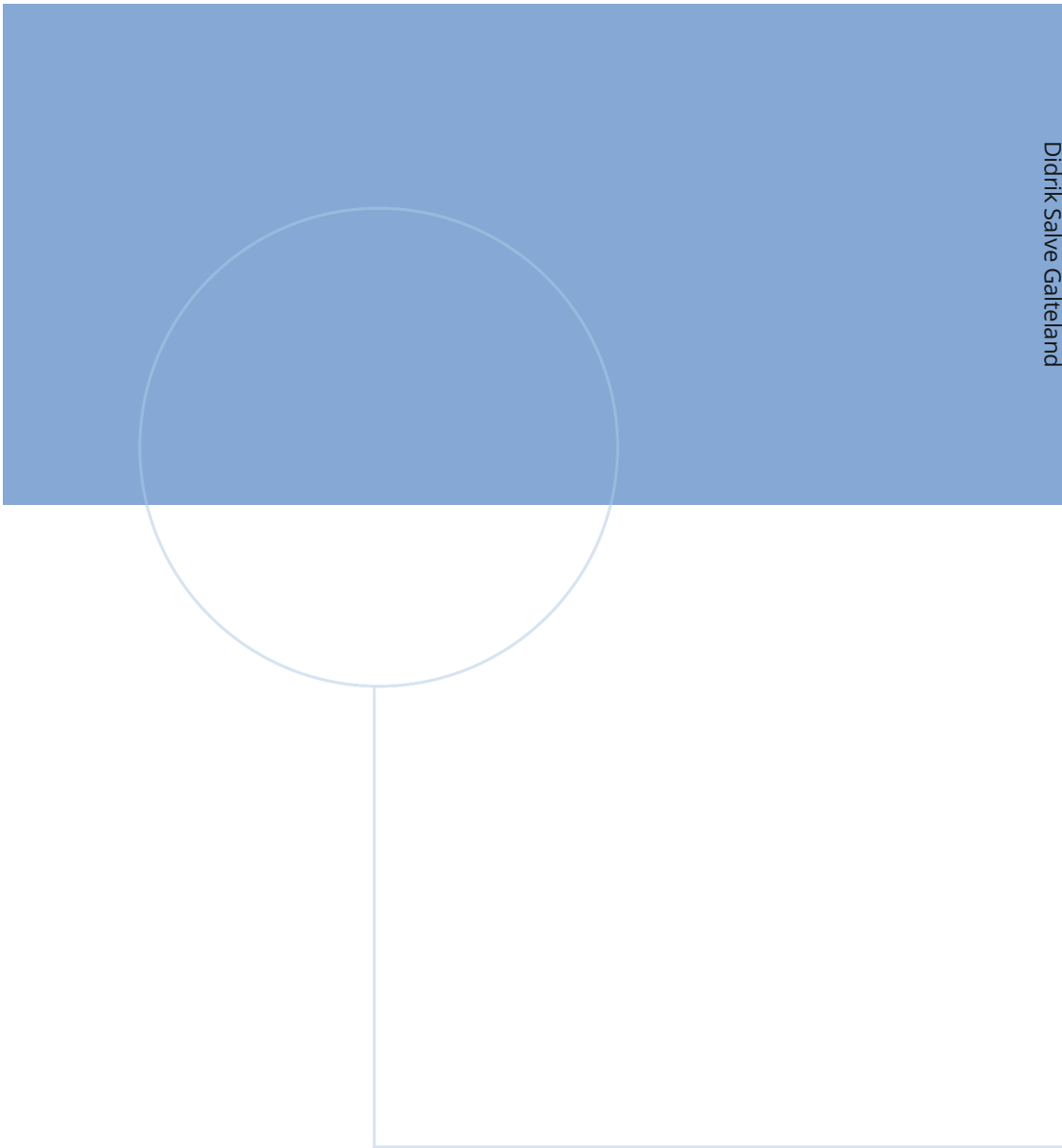
[65] H. Li, A. Gordon, H. Zhao, V. Casser, and A. Angelova, "Unsupervised Monocular Depth Learning in Dynamic Scenes," *arXiv e-prints*, arXiv:2010.16404, arXiv:2010.16404, Oct. 2020. arXiv: 2010.16404 [cs.CV].

[66] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, "Emerging Properties in Self-Supervised Vision Transformers," *arXiv e-prints*, arXiv:2104.14294, arXiv:2104.14294, Apr. 2021. arXiv: 2104.14294 [cs.CV].

[67] C. Meisel, R. El Atrache, M. Jackson, S. Schubach, C. Ufongene, and T. Loddenkemper, "Deep learning from wristband sensor data: towards wearable, non-invasive seizure forecasting," *arXiv e-prints*, arXiv:1906.00511, arXiv:1906.00511, Jun. 2019. arXiv: 1906.00511 [q-bio.NC].

[68] J. Cohen, *Computer vision at tesla*, Jun. 2021. [Online]. Available: https://heartbeat.fritz.ai/computer-vision-at-tesla-cd5e88074376.

[69] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5122–5130. DOI: 10.1109/CVPR.2017.544.



Didrik Salve Galteland

NTNU
Norwegian University of
Science and Technology